INFORMATION TO USERS

This was produced from a copy of a document sent to us for microfilming. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help you understand markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure you of complete continuity.

2. When an image on the film is obliterated with a round black mark it is an indication that the film inspector noticed either blurred copy because of movement during exposure, or duplicate copy. Unless we meant to delete copyrighted materials that should not have been filmed, you will find a good image of the page in the adjacent frame. If copyrighted materials were deleted you will find a target note listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed the photographer has followed a definite method in "sectioning" the material. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again--beginning below the first row and continuing on until complete.

4. For any illustrations that cannot be reproduced satisfactorily by xerography, photographic prints can be purchased at additional cost and tipped into your xerographic copy. Requests can be made to our Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases we have filmed the best available copy.
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark.

1. Glossy photographs or pages ______
2. Colored illustrations, paper or print ______
3. Photographs with dark background ______
4. Illustrations are poor copy ______
5. Pages with black marks, not original copy ______
6. Print shows through as there is text on both sides of page ______
7. Indistinct, broken or small print on several pages ______
8. Print exceeds margin requirements ______
9. Tightly bound copy with print lost in spine ______
10. Computer printout pages with indistinct print ______
11. Page(s) ______ lacking when material received, and not available from school or author.
12. Page(s) ______ seem to be missing in numbering only as text follows.
13. Two pages numbered ______. Text follows.
14. Curling and wrinkled pages ______
15. Other
DESIGN AND ANALYSIS OF A MULTI-BACKEND DATABASE SYSTEM FOR PERFORMANCE IMPROVEMENT AND CAPACITY GROWTH

DISSERTATION

Presented in partial fulfillment of the requirements for the degree Doctor of Philosophy in the Graduate School of the Ohio State University

By

M. Jaishankar Menon, B.Tech, M.S.

The Ohio State University
1981

Reading Committee:

Professor D. K. Lieu
Professor J. T. Liu
Professor D. S. Kerr

Approved by

Adviser

Department of Computer and Information Science
To My Parents
ACKNOWLEDGEMENTS

I am deeply indebted to my adviser, Professor David K. Hsiao of the Department of Computer and Information Science for his untiring support and constant encouragement throughout the course of this research. I can never thank him enough for moulding the ideas in this dissertation into the concrete form that they now take. Nor can I thank him enough for immeasurably improving my writing style. I would also like to thank Dr. Fred Maryanski and Professors Douglas S. Kerr and Ming-Tsan Liu for reading my dissertation carefully and for suggesting some changes which lead to better readability.

I would like to acknowledge with gratitude the support provided by the Office of Naval Research for this research effort under contract N00014-75-C-0573 and the research equipment provided in the Laboratory for Database Systems Research which is jointly funded by Digital Equipment Corporation, Office of Naval Research and Ohio State
University.

I wish to thank Tamer Ozsu for being a very good friend and office-mate and for reading an earlier draft of this dissertation. Words cannot repay my debt to him.

Ahmed El-Kagarmid has my special thanks for being the best friend and companion that a fellow could have. His friendship helped to lighten the burden of research to no small extent. Once again, the friendship goes beyond mere words.

Finally, I would like to acknowledge with gratitude Dilip Soni who has been a good friend through my entire four years.
VITA

August 9, 1956
Born, Ottapalam, India

1977
B Tech. Electrical Engineering
Indian Institute of Technology
Madras, India

1978
M S., Computer and Information
Science, Ohio State University,
Columbus, Ohio

1978-1981
Graduate Research Associate
Department of Computer and
Information Science
The Ohio State University
Columbus, Ohio

PUBLICATIONS

"The Access Control Mechanism of a Database Computer (DBC)",
co-authored with D.K. Hsiao, Proceedings of the Fifth
Annual Workshop on Computer Architecture for Non-numeric
Processing, Pacific Grove California, March 11-14, 1980.

"The Impact of Auxiliary Information and Update Operations
on Database Machine Architecture", co-authored with D.K.
Hsiao, International Congress on Applied Systems Research

"Design and Analysis of Relational Join Operation for VLSI
Implementation", co-authored with D.K. Hsiao, Proceedings
of the Seventh International Conference on Very Large
Databases, Cannes, France, 1981.
FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in Computer Systems Programming:
  Professor David K. Hsiao

Studies in Computer Architecture:
  Professor Ming Tsan Liu

Studies in Programming Languages:
  Professor N. Soundararajan
# Table of Contents

**DEDICATION** ......................................................... ii

**ACKNOWLEDGEMENTS** ................................................ iii

**VITA** ........................................................................... v

**LIST OF TABLES** ........................................................ xix

**LIST OF FIGURES** ........................................................ xx

**Chapter**

I. **INTRODUCTION** ....................................................... 1
   1.1 The Goal ............................................................... 2
   1.2 A Taxonomy of Existing Systems ................................. 3
   1.3 Design Issues to Be Studied ..................................... 12
      1.3.1 Hardware Issues ................................................ 13
         A. The Back-end Interconnection ................................. 13
         B. The Database Store Interconnection ....................... 14
      1.3.2 System Issues .................................................. 16
         A. The Database Placement ........................................ 16
         B. The Execution Mode ............................................ 16
         C. Directory Structure ............................................ 17
         D. The Directory Placement ..................................... 17
         E. The Security Issue .............................................. 18
         F. The Data Model .................................................. 18
         G. The Data Manipulation Language ............................. 18
         H. The Reliability Issue .......................................... 18
1.3.3 Software Issues

A. The Degree of Concurrency

B. Consistency Control and
   Deadlock Avoidance

1.4 Terminology

1.5 Contributions of Research

II. A SURVEY OF TYPICAL SYSTEMS AND A STUDY OF
    SYSTEM ISSUES FOR DESIGN DECISIONS

2.1 A Survey of Typical Software-Oriented,
    Multiple Back-ends

2.1.1 RDBM - A Relational Database System
   A. The Problem of Channel Limitation
   B. The Problem of Software Specialization
   C. The Problem of Controller Limitation
   D. The Problem of Data Model Limitation

2.1.2 DIRECT - A Multiple Back-end Relational System
   A. The Problem of Hardware Specialization
   B. The Problems of Control Message Traffic and Controller Limitation
   C. The Problem of Multiple Request Execution
   D. The Problem of Data Model Limitation

2.1.3 Stonebraker's Machine - A Distributed Database System
   A. The Problem of Back-end Limitation
   B. The Problem of the Specialized Back-end
   C. The Problem of Controller Limitation
D. The Problem of Multiple Request Execution.................. 49
E. The Problem of Device Limitation......................... 49
F. The Problem of Control Message Traffic...................... 50
G. The Problem of Data Model Limitation....................... 50

2.1.4 DBMAC - An Italian Database System..................... 50
A. The Problem of Channel Limitation......................... 52
B. The Problem of Software Specialization..................... 53
C. The Problem of Back-end Limitation......................... 53
D. The Problem of Data Model Limitation....................... 54

2.2 Basic Design Considerations for a Multi-Mini Database System (MDBS). .... 54
2.2.1 Nine Design Goals........................................ 55
2.2.2 Towards an Ideal System Architecture..................... 56

2.3 First Design Decision - Eliminating the Channel, Back-end and Device Limitation Problems........................................ 58
2.3.1 The Need of a Data Placement Strategy.................... 62
2.3.2 An Evaluation of Data Placement Strategies................ 65
2.3.3 An Evaluation of Data Placement Strategies Using More Refined Assumptions.......................... 75
2.3.3.1 The Choice of a Superior Data Placement Strategy on the Basis of Better Response Time.......... 78
2.3.3.2 The Choice of a Superior Data Placement Strategy on the Basis of Better Storage Utilization..... 84

ix
2.3.4 Next Step in the Design Process................. 86

2.4 Second Design Decision - Minimizing the Problem of Control Message Traffic.............. 88

2.4.1 The Need for a Broadcast Capability............................... 90

2.4.2 An Evaluation of the Broadcast Capability With More Refined Assumptions............. 92

2.5 An Overview of the MDBS Architecture and Design.................................. 99

III. THE CHOICE OF A DATA MODEL AND A DATA MANIPULATION LANGUAGE....................... 103

3.1 Three Selection Criteria.................. 103

3.1.1 The Translation Criterion............... 104

3.1.2 The Partition Criterion............... 105

3.1.3 The Language Criterion.................. 111

3.2 The Attribute-Based Model................. 112

3.2.1 Concepts and Terminology............ 115

3.2.2 The Data Manipulation Language (DML).................................. 120

A. Retrieve........................................ 120

B. Insert......................................... 122

C. Delete........................................ 122

D. Update........................................ 123

3.2.3 The Notion of a Transaction............ 125

3.2.4 Basic Requests vs. Aggregate Requests................................. 129

IV. THE PROCESS OF REQUEST EXECUTION............ 131

4.1 The Notion of Record Clusters................. 134

4.1.1 Cluster Formation......................... 136

4.1.2 An Example of Cluster Formation........ 140

4.1.3 Clusters Determination During Request Execution............................. 144

4.1.4 An Example of Clusters Determination During Request
4.2 Directory Management

4.2.1 Two Phases of Processing
- Descriptor Processing and Address Generation

4.2.2 Processing Strategies for Multiple Back-ends

A. The Centralized Strategy
B. The Partially Centralized Strategy
C. The Rotating Strategy
D. The Rotating Without Controller Strategy
E. The Fully Duplicated Strategy
F. The Descriptors Dividing by Attribute Strategy
G. The Descriptors Division Within Attribute Strategy
H. The Fully Replicated Strategy

4.2.3 Performance Evaluation of the Directory Management Strategies

A. Time Analyses and Performance Equations
B. Computations and Their Interpretations Resulting from the Performance Equations
C. A Preliminary Conclusion Based on the Performance Equations
E. Performance Analysis Based on a Closed Queueing Network Model
   E.1 The I/O Submodel for Single Requests
   E.2 The I/O Submodel for Bulk Requests with Fixed Bulk Size
   E.3 The I/O Submodel for Bulk Requests with Variable
F. Modelling the Eight Strategies for Evaluation

F.1 The Centralized Model
F.2 The Partially Centralized Model
F.3 The Rotating Model
F.4 The Rotating Without Controller Model
F.5 The Fully Duplicated Model
F.6 The Descriptors Dividing By Attribute Model
F.7 The Descriptors Division Within Attribute Model
F.8 The Fully Replicated Model

G. Results of the Queueing Network Modelling of Strategies

4.2.4 Storage Requirements of Directory Management Strategies

A. Size Estimation of the Descriptor-to-Descriptor-Id Tables (DDITs)
B. Size Estimation of the Augmented Cluster Definition Table (CUT)
C. Interpretations of the Results on Sizes

4.3 The Entire Process of Request Execution

4.3.1 Executing a Retrieve Request
4.3.2 Executing a Delete Request
4.3.3 Executing an Update Request
4.3.4 Executing an Insert Request

V. THE PROCESS OF REQUEST EXECUTION WITH ACCESS CONTROL

A. The Authorization Step
B. Three Types of Access Control - by Granules, by Statistics and by Values
C. A New Mode of Operation - Precision Control by Multiple Back-ends............. 265

5.1 Access Control as Exercised by the Database Creator......................... 266
5.2 Determination and Organization of the Exact Access Control from the Database-Creator-Specified Information................................. 274

5.2.1 Controlling Access to Authorized Clusters......................... 275
5.2.2 Organizing and Storing Cluster Control Tables......................... 277

5.3 Request Execution With Fine Granularity of Access Control............. 282

5.3.1 Executing Insert Requests..... 284
5.3.2 Executing Non-Insert Requests. 285

A. A Case of Compromised Access Control Due to Alternative Operation.... 286
B. Two More Cases of Compromised Access Control Due to Alternative Operation......................... 289

5.3.3 The 'Conservative' and Precision Access Control Mechanism............. 295
5.3.4 An Example of the Process of Request Execution With Access Control......................... 298

5.4 New Capabilities of the Access Control Mechanism......................... 300

5.4.1 Statistical Access Control..... 300

A. An Example of Compromised Access Control Due to Users Own Aggregate Operations... 304
B. An Expanded Procedure for Effective Statistical Access Control............. 305
5.4.2 Value-Dependent Access Control

5.5 Access Control for Transactions of Multiple Requests

5.5.1 Stand-Alone Execution of Transactions

5.5.2 Attached Execution of Transactions

5.5.3 The Choice of Transaction Execution for Access Control

5.6 The Management of Access Control Information

5.6.1 Denying a User from Any Access to the Database

5.6.2 Adding a New User of the Database

5.6.3 Changing the User's Access Operations Only

5.6.4 Changing the User's Permitted Data Granules

A. Defining New Granules With New Descriptors

B. Creating Larger Granules By Deleting Descriptors

C. Creating Larger Granules By Coalescing Descriptors

D. Defining New Granules By Splitting Descriptors

VI. CONCURRENCY CONTROL FOR MULTIPLE BACK-ENDS AND CONSISTENCY OF PARTITIONED DATABASES

6.1 Is There a Necessity for Concurrency Control?

A. The Throughput Issue

B. The Response Time Issue

C. The Multiple Disk Drive Issue

6.2 What are the Necessary and Sufficient Conditions for a Consistent Partitioned Database Utilizing Multiple Back-ends?
6.3 Monolithic Consistency and Non-permutable Requests
6.4 Notations and Terminology
6.5 Request Permutabilities
6.6 Request Compatibilities
6.7 Cluster-Based Permutabilities and Compatibilities

6.7.1 Determining the Set of Future Clusters for an Update Request

A. Determining the Future Cluster of a Record to be Updated Without Having Seen the Record

B. Determining All the Future Clusters in Order to Lock Them Up For Record Updates

C. Determining Incompatible and Non-Permutable Requests

6.7.2 A Case of 'Over-Determination'

6.8 The Cluster-Based Concurrency Algorithm

6.8.1 Incompatible and Non-permutable Locks

6.8.2 The Execution Sequence of a Secure Transaction

6.8.3 The Concurrency Control Mechanism

6.9 An Examination of the Concurrency Control Mechanism

6.9.1 New Solutions for Centralized Database Concurrency Control

6.9.2 New Solutions for Partitioned Database Concurrency Control

A. Rich Semantics in DML and New Concept of Permutability

B. Better Throughput and Lower Control Message
Traffic.................. 373

C No Back-end Limitation Problem.................. 374

D. A Question of Overhead Incurred During Concurrency Control. .............. 375

E. Free From Starvation Errors.................. 375

E.1 Transaction Execution by the MDBS Solution.... 377

E.2 Transaction Execution by the Starvation Solution........... 378

6.10 The Execution of Incompletely-Specified Transactions............... 379

6.10.1 Problems With Backing Up Transactions............. 381

6.10.2 The No-Back-Up Solution ....... 382

6.10.3 A Solution with Backing Up........ 383

VII. DESIGN AND PERFORMANCE ANALYSIS.......... 388

7.1 A Simulation Model of MDBS............ 390

7.1.1 Sequence of Events for a Retrieve Request.............. 391

A. The Parsing Phase.............. 391

B The Descriptor Search Phase. 392

C. The Address Generation Phase Including Access Control... 393

D. The Secondary Memory Retrieval Phase............. 394

E. The Response Phase............. 395

7.1.2 Sequence of Events for a Delete Request.............. 395

A. The Tag-for-Deletion Phase. 395

B. The Acknowledgement Phase... 396

7.1.3 Sequence of Events for an Update Request.............. 396

A. The Record Modification
7.1.4 The Sequence of Events for an Insert Request

- A. The Parsing Phase
- B. The Descriptor Search and Initial Address Generation Phase
- C. The Back-end Selection Phase
- D. The Record Insertion Phase
- E. The Acknowledgement Phase

7.2 Simulation Environments and A Measure of Performance

- 7.2.1 Retrieve-Intensive vs. Update-Intensive
- 7.2.2 Cluster Size vs Request Size
- 7.2.3 Hardware Configurations and Requirements
- 7.2.4 A Measure of Anticipated Performance vs. Ideal Performance

7.3 MDBS Performance Under Various Conditions

- 7.3.1 Intensive Retrieval Involving Large Clusters
- 7.3.2 Intensive Retrieval Involving Small Clusters
- 7.3.3 Intensive Update Involving Large Clusters
- 7.3.4 Intensive Update Involving Small Clusters
- 7.3.5 Effects of Broadcasting on Performance
- 7.3.6 Three Observations of Strong Design and Performance Factors - High-Volume Processing, Intensive Update and Inexpensive Broadcast Bus

7.4 A More Refined Simulation of MDBS...
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. The Response Time (in msecs) of MDBS Under Various Data Placement Strategies</td>
<td>70</td>
</tr>
<tr>
<td>2. The Improvement Caused by a Good Placement Strategy</td>
<td>71</td>
</tr>
<tr>
<td>3. Response Time Results for the Various Strategies for Small-Sized Requests</td>
<td>80</td>
</tr>
<tr>
<td>4. Response Time Results for the Various Strategies for Medium-Sized Requests</td>
<td>81</td>
</tr>
<tr>
<td>5. Response Time Results for the Various Strategies for Large-Sized Requests</td>
<td>82</td>
</tr>
<tr>
<td>6. Comparison of Storage Wastage Between Two Different Placement Policies</td>
<td>87</td>
</tr>
<tr>
<td>7. Response Time for MDBS with and without Broadcast Facility</td>
<td>91</td>
</tr>
<tr>
<td>8. Comparing MDBS with and without Broadcast for Small-Sized Requests</td>
<td>93</td>
</tr>
<tr>
<td>9. Comparing MDBS with and without Broadcast for Medium-Sized Requests</td>
<td>94</td>
</tr>
<tr>
<td>10. Comparing MDBS with and without Broadcast for Large-Sized Requests</td>
<td>95</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>A Taxonomy of Database Management Systems</td>
<td>4</td>
</tr>
<tr>
<td>2.</td>
<td>Configurations of Back-ends and Disk Drives</td>
<td>15</td>
</tr>
<tr>
<td>3.</td>
<td>RDBM - A Relational Database System</td>
<td>31</td>
</tr>
<tr>
<td>4.</td>
<td>The DIRECT System</td>
<td>38</td>
</tr>
<tr>
<td>5.</td>
<td>Stonebraker's Machine - A Distributed Database System</td>
<td>47</td>
</tr>
<tr>
<td>6.</td>
<td>A View of the Italian Database System</td>
<td>51</td>
</tr>
<tr>
<td>7.</td>
<td>An Overview of MDBS Architecture</td>
<td>61</td>
</tr>
<tr>
<td>8.</td>
<td>Data Placement for 6 Records</td>
<td>63</td>
</tr>
<tr>
<td>9.</td>
<td>Three Different Data Placement Strategies</td>
<td>66</td>
</tr>
<tr>
<td>10.</td>
<td>A Sample Network Database With Four Record Types and Three Set Types</td>
<td>106</td>
</tr>
<tr>
<td>11.</td>
<td>Partitioning the Database of Figure 10 on three back-ends</td>
<td>109</td>
</tr>
<tr>
<td>12.</td>
<td>A Sample Database of Two Files (Adopted from [Eswa76])</td>
<td>127</td>
</tr>
<tr>
<td>13.</td>
<td>A Database of Two Files and its Clustering Descriptors</td>
<td>141</td>
</tr>
<tr>
<td>14.</td>
<td>The Descriptor-to-Descriptor-id Table (DDIT)</td>
<td>142</td>
</tr>
<tr>
<td>15.</td>
<td>The Cluster Definition Table (CDT)</td>
<td>143</td>
</tr>
</tbody>
</table>


19. Closed Queueing Network Model of MDBS

20. Queueing Model of a Single Channel Disk System

21. Queueing Model of a Disk System with Bulk Arrivals of Size T at a Rate L

22. Queueing Model of a Disk System with Bulk Arrivals of Variable Bulk Size

23. Queueing Network Model Results for Strategy A

24. Queueing Network Model Results for Strategy B

25. Queueing Network Model Results for Strategy C

26. Queueing Network Model Results for Strategy D

27. Queueing Network Model Results for Strategies E and F

28. Queueing Network Model Results for Strategy G

29. Queueing Network Model Results for Strategy H

30. MDBS Response Times Under Various Directory Management Strategies
<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>31. Directory Size (in kbytes) for Various Directory Management Strategies</td>
<td>245</td>
</tr>
<tr>
<td>32. The Cluster-Id-To-Next-Back-end Table (CINBT)</td>
<td>259</td>
</tr>
<tr>
<td>33. A Sample Database for Illustrating Field-Level Access Control</td>
<td>271</td>
</tr>
<tr>
<td>34. The Augmented Descriptor-to-Descriptor-Id Table (DDIT) for the Sample Database of Figure 33</td>
<td>273</td>
</tr>
<tr>
<td>35. The Cluster Definition Table (CDT)</td>
<td>276</td>
</tr>
<tr>
<td>36. The Augmented Cluster Definition Table for User 2</td>
<td>279</td>
</tr>
<tr>
<td>37. The Augmented CDT for the Sample Database of Figure 33</td>
<td>280</td>
</tr>
<tr>
<td>38. The Descriptors and Corresponding Sets of Field-level Access Controls Formed for the Database of Figure 33</td>
<td>310</td>
</tr>
<tr>
<td>39. The Augmented DDIT of Figure 33 Before and After the Addition of User 3 to the System</td>
<td>317</td>
</tr>
<tr>
<td>40. The Augmented DDIT After Changing the Set of Field-Level Access Controls for User 2 on Descriptor D1</td>
<td>319</td>
</tr>
<tr>
<td>41. The New Separate Cluster Definition Table (SCDT) for User 2</td>
<td>321</td>
</tr>
<tr>
<td>42. The Lock Table</td>
<td>356</td>
</tr>
<tr>
<td>43. Three Algorithms for Cluster Queue Management</td>
<td>362</td>
</tr>
</tbody>
</table>
44. The MDBS Response Times in a Retrieve-Intensive Environment with Large Amount of Data Involved. ......................405

45. The MDBS Response Times in a Retrieve-Intensive Environment with Small Amount of Data Involvement .........................407

46. The MDBS Response Times in an Update-Intensive Environment Involving Large Clusters ..................................................410

47. Intensive Update Involving Small Clusters ..................412

48. The Response Times of MDBS Effected by the Broadcast Bus Speeds .................................................................415

49. The Response Times of MDBS Under a Refined Policy Simulation .................................................................418
For solving database management problems, many researchers [Babb79, Bane78, Cope73, Schu79, Wah80] have proposed using special-purpose hardware, known as database machines. However, it has not yet been demonstrated that these database machines are cost-effective. A different approach to the solution of database management problems may involve the use of conventional hardware elements, perhaps in large number, with most of the database management functions carried out in software. In other words, we may be searching for hardware solutions without having exhaustively considered all possible software solutions to database management problems. In this dissertation, we
emphasize the software approach to the solution of database management problems.

1.1 The Goal

The goal is to investigate whether it is possible to use a multiplicity of general-purpose processing and storage elements (specifically, minicomputers and disk drives), novel hardware configuration and innovative software design for achieving throughput gain and response time improvement over conventional database management systems. Ideally, the performance gains and improvements should be proportional to the multiplicity of processing elements used. If this ideal goal cannot be achieved, then the case for database machines that use special-purpose hardware becomes very strong. On the other hand, if it can be shown that such an ideal system can be obtained, then a cost-effective way for high-volume and great-capacity database management may become more readily available. Database machines may still provide better performance, but they will be more expensive. Furthermore, database machines may represent a distant solution whose time is not yet near.
In this dissertation, we will propose a hardware configuration of multiple minicomputer systems and a software database management system for the configuration. We will use simulation studies to see how closely the ideal goal has been achieved by our proposed configuration and design.

1.2 A Taxonomy of Existing Systems

One way to give the correct perspective to our work is by way of a taxonomy of database systems and machines. By developing the taxonomy and indicating the relative position of our work within this taxonomy, we may also explain the similarities and differences of our work with that of others which are either operational or being proposed. Finally we will show the advantages of the software approach over the machine approach. The taxonomy we developed is shown in Figure 1.

At the highest level, we differentiate between systems which utilize a central controller to simplify the control functions and those which do not utilize a central controller. In general, it is the case that systems which do not utilize a central controller are those where the database is geographically dispersed and the various
Figure 1. A Taxonomy of Database Management Systems
computers have to be connected together by a network. Examples of such systems are SDD-1 [Roth80], Distributed Ingres [Ston76a], and Muffin [Ston79]. Also, generally speaking, systems that do not use a central controller will have either partially or fully duplicated databases. The need for having duplicate databases becomes important in a geographically dispersed database where data transfers among computers via a network are expensive. Hence, it is important to duplicate data so that the data is very close to the point where it is actually needed. In systems with duplicate databases, the concurrency control algorithms are mostly complex and require large numbers of messages to be exchanged [Hsi81]. This dissertation is not concerned with systems that do not utilize a central controller and where the databases are duplicated, though this could very well be an area for future research.

In developing our taxonomy further, we divide the systems with central controllers into two classes, namely, the hardware-oriented systems and the software-oriented systems. Hardware-oriented systems (also called database machines) are those which typically use special-purpose hardware to perform a large number of the database management functions. These systems [Bane78, Cope73, Schu79] process the data 'on the fly' while the data is being read from the disk tracks. Special-purpose processors
which are associated with blocks of secondary storage are utilized to perform on-the-fly processing. Our taxonomy is, of course, subjective in defining a hardware-oriented system in terms of the number of the database management functions being accomplished in hardware. What is a 'large' number of the database management functions? Clearly, there will be some systems which fall in the borderline between hardware-oriented and software-oriented systems. Note that we do not consider a database management system like DIRECT [Dewi78] to be a hardware-oriented system even though it is often referred to as a database machine in the literature. This is because the only special-purpose hardware in DIRECT is a crossbar switch. All the database management functions in DIRECT are performed by software in conventional processors. Hence, by our classification, DIRECT is a software-oriented database management system.

Software-oriented systems are those which do not use a significant amount of special-purpose hardware and where most of the functions of database management are done in software. Such systems may be further subdivided into three categories as conventional database management systems, single back-end database management systems and multiple back-end database management systems. In a conventional database management system, all of the major software components, such as the operating system, the database
management system and user (application) programs, are executed on a single computer which has direct access to the database stored in the secondary memory. Performance upgrades in a conventional database management system are usually costly and disruptive. For instance, it may be necessary to add large memory modules or to incorporate more channels or to replace the central processor with a more powerful model. Such upgrades often require major effort so that it is well accepted that conventional database management systems are not easily extensible. In this context, we define extensibility of a database management system as the ability to upgrade it with

1. no modification to existing software,
2. no additional programming,
3. no modification to existing hardware and
4. no major disruption of system activity when additional hardware is being incorporated into the existing hardware.

A new configuration for database management called the back-end configuration was proposed in [Cana74]. Excellent descriptions of the back-end concept and its advantages and disadvantages are given in [Lowe76, Mary80] and will not be repeated here. Briefly, this configuration utilizes two computer systems -- a host and a back-end. The database management functions are implemented on the back-end which
has exclusive access to the database on secondary storage devices. We shall refer to such a system as a single back-end database management system, since multiple back-end systems have also been proposed [Lowe76, Ston78, Dewi78]. Among the advantages claimed of single back-end database management systems is that performance upgrades are less disruptive, more manageable and on a smaller scale than in a conventional database management system. First, upgrading the back-end requires no modification to application programs since they are executed in the host. Furthermore, the back-end separates the characteristics of the secondary storage devices from the characteristics of the host. Thus, new storage technology may be employed without changing hardware or software in the host. However, single back-ends have the disadvantage that, ultimately, performance upgrades will require replacement of the back-end and this may entail software modifications and hardware disruption.

The next logical step in the evolution of database management systems is the multiple back-end system, and this has been suggested by a number of researchers [Dewi78, Lowe76, Ston78]. This approach employs multiple computer systems for database management with a software-oriented system and a centralized controller. A few words regarding the differences between our system and other software-oriented multiple back-end systems with controllers
[Dewi78, Lowe76, Ston78] is now in order. The work of [Lowe76] did not represent the design of any specific database management system. That work was essentially in the nature of an idea for an architecture which others could utilize to design systems. The works of [Ston78, Dewi78], however, are actual designs. The difference between our work and that of these two researchers are explained at a very detailed level in Chapter II. Here, we satisfy ourselves with some differences in terms of overall objectives. None of the above-mentioned researchers emphasize the extensibility of the multiple back-end approach nor do they seem very concerned about designing their respective multiple back-end systems to make them extensible. In [Lowe76] for example, the multiple back-end system is considered from the viewpoint of enhanced reliability over single back-end systems. In [Dewi78, Ston78], however, the emphasis is on the fact that multiple back-ends can provide greater throughput than single back-end systems. However, we believe in more specific findings. In other words, we believe that there are designs in which the throughput can be proportional to the multiplicity of back-ends. None of the above-mentioned systems has revealed such a design. We, on the other hand, reveal a design of a multiple back-end system which could achieve the ideal goal of response time being inversely
proportional to the multiplicity of back-ends. Of course, it is not possible for this inverse proportionality to hold when the number of back-ends is extremely large. However, we believe that present system designers have overlooked certain crucial system bottlenecks so that this inverse proportionality does not hold in their systems even for very low number of back-ends. Careful design of a system will allow this inverse proportionality to hold for reasonably large number of back-ends. It is also our belief that our multiple back-end system is extensible. That is, new storage devices and back-ends may be added to the configuration to improve its storage and performance capabilities without the need for re-designing and re-programming of the software and without major disruption of the existing hardware. We will be interested in multiple back-ends from the viewpoint of extensibility to improve response time and throughput. Furthermore, we will not utilize any special-purpose hardware in our system.

As a result of the extensible nature of our system, we make the following claim about our system. Consider a user of our database management system who finds that the system is saturated due to database growth and transaction increase. The user would like to upgrade the system. The designers of database machines and conventional database management systems would offer the user the alternative of
using their database machines and their systems, respectively, thereby replacing the existing software and hardware. Instead, we offer the user a different alternative. We do not ask the user to replace the existing software, since the existing software is extensible and requires only a system generation. We also do not ask the user to replace the existing hardware. We merely ask the user to add identical hardware; consequently when and if the need to upgrade the system arises, the user simply adds more hardware and uses the same existing software for performance gain and capacity growth.

The presentation of the taxonomy and of the place of our system in that taxonomy has made clear some of the advantages and unique characteristics of the software approach to database management system design which we intend to follow. Two other considerations confirm that this may be a viable alternative to follow. We may term these as hardware considerations and software considerations. Both of these are considered in terms of our software approach and the database machines approach. First of all, from a hardware point of view, the conventional processors and disk drives will be cheaper than special-purpose processors and disk drives which may be needed in a database machine. From a software point of view, software for a large number of processors may be
needed in this approach. However, novel software design may be used to ensure that the software in each of the minicomputers is identical. Thus, the software complexity is not proportional to the multiplicity of minicomputers. Furthermore, we are trying to support large databases that evolve and grow over time. This growth process is generally gradual. However, performance upgrades are necessary from time to time for such database growth. We have already indicated that our particular approach to database management systems can lead to extensible systems and improved performance to accommodate the growth and the evolution. Thus, from a software, hardware and database growth point of view, the software approach using multiple back-ends seems preferable to the database machines approach. Thus, such an approach is certainly worth investigating and that is the focus of this dissertation.

1.3 Design Issues To Be Studied

Before an extensible system can be developed, a number of issues related to multiple back-end systems must be resolved. None of the researchers mentioned above have provided solutions to all these issues, though some researchers have proposed solutions to a subset of them. Let us enumerate below the various issues to be considered
in the design of a multiple back-end system. The issues may be divided into three broad categories as hardware issues, system issues and software issues.

1.3 1 Hardware Issues

A. The Back-end Interconnection

The questions to be resolved here are, "What is the optimal way of interconnecting the back-ends together?", and "What is the optimal way of connecting the host to the back-ends?" In answering these questions, we must recall that the throughput of multiple processor systems increases significantly only for the first few additional processors. At some point, the throughput actually begins to decrease with each additional processor. Examples of this phenomenon are documented by [Chu80] and [Jenn77]. This decrease in throughput is due to excessive interprocessor communication during the execution of a single task which causes a saturation effect [Chu78]. We must try to avoid this excessive interprocessor communication in designing an optimal multiple back-end system.
B. The Database Store Interconnection

Another issue is the question of which secondary devices should be connected to which back-ends. For instance, if two back-ends and two disk drives are given, would the configuration where each back-end is connected to exactly one disk drive be superior to the configuration where each back-end is connected to both disk drives (see Figure 2)? The latter configuration is more flexible since any back-end may be employed to execute any user request. However, it is more expensive in terms of hardware complexity. Furthermore, if two back-ends can access the same data, there exists the problem of deciding which back-end should be allowed to access a particular data item. Such a configuration also complicates the concurrency control issue, since one back-end may now read and update the same data being read and possibly updated by another back-end.
Figure 2a. A Configuration Where Each Back-end is Connected to Only Certain Disk Drives

Figure 2b. A Configuration Where Each Back-end is Connected to all Disk Drives

Figure 2. Configurations of Back-ends and Disk Drives
1.3.2 System Issues

A. The Database Placement

Here, the issue to be resolved is regarding the best way of placing the files constituting the database across the various back-ends in order to achieve the maximum amount of parallel access for the system during read and write operations. In other words, how should the database be partitioned? As much as possible, we should try to ensure that the records constituting the response set to a user request are not stored at a single back-end. Rather, these records in the response set must be distributed across the various back-ends. Such a placement policy should lead to maximal parallel access. Techniques for achieving such a placement policy must be found.

B. The Execution Mode

We ask whether the multiple back-ends should execute in a single-instruction-stream-multiple-data-stream (SIMD) fashion or in a multiple-instruction-stream-multiple-data-stream (MIMD) fashion. That is, should each of the back-ends be executing the same request but on different data (SIMD), or should the back-ends be executing in an asynchronous fashion (MIMD).
The results of our study in [Meno80] have shown us that MIMD configurations need not always be superior.

C. Directory Structure

Many database management systems also have a secondary body of information (often referred to as the directory) which is used to decrease the search space (and therefore search time) of the database. Issues to be resolved here are whether a directory is indeed necessary and if it is necessary, then what form should it take? Should we use the inverted list organization, the multilist organization, or any of the whole spectrum of alternatives as elucidated in [Hsia70]?

D. The Directory Placement

Having decided on the nature of the directory, we next need to answer the following question. What is the best place to store the directory? Should it be stored at the host, in one of the back-ends, or in all of the back-ends? Should it be duplicated or partitioned? The tradeoffs to be considered are those of storage requirements versus reliability, system throughput and response time.
E. The Security Issue

An important issue that must always be considered in the design of a database system is how security is to be enforced. That is, the question of deciding, in a multi-user environment, who should have what access to which data.

F. The Data Model

Any database management system must decide what data model, e.g., relational, network and hierarchical, it will support. This decision must be made irrespective of whether we have a conventional system, a single back-end system or a multiple back-end system. However, it is possible that some issues peculiar to multiple back-end systems may impact our choice of a data model. For example, it will be shown that it is easier to partition the database if it is represented in one kind of data model and not in the other kind. The partitioning criteria is relevant only in the context of multiple back-end database management systems.

G. The Data Manipulation Language

We must also decide upon a language which can be used by the users to manipulate the database in an easy fashion. The choice of a data manipulation language will be closely
related to the chosen data model.

H. The Reliability Issue

Finally, it is necessary to make the system as reliable as possible. Thus, we would like to design a system which continues to perform the database management functions (perhaps in a degraded mode) in spite of the loss of one or more back-ends.

1.3.3 Software Issues

A. The Degree of Concurrency

We are designing an architecture that supports multiple users concurrently. At each back-end, we have the choice of processing the requests from these multiple users in an interleaved manner or one at a time. If the requests are not interleaved, then the software in the back-end is simplified. However, the price to be paid in terms of increased user response time and low back-end utilization may be too high. For this reason, we may want to support concurrent request processing at each back-end.
B. Consistency Control And Deadlock Avoidance

If each back-end is to handle multiple requests and multiple transactions, then algorithms must be developed to ensure consistency of the database in the presence of multiple transactions. These algorithms must ensure that each transaction will behave as if it were the only one in the system. Consistency control algorithms may or may not permit deadlocks to occur. If deadlocks are permitted to occur, other algorithms must be developed for detecting and recovering from deadlocks.

In the design of our multiple back-end system, we will adequately cover every issue except the one on reliability. The problem of making our system reliable is left as a topic for future research and will not be addressed in this dissertation.

1.4 Terminology

We wish to end this section with a brief look at terminology. Throughout the remainder of this report, we will refer to the multiple back-end system which we are attempting to design as the multi-mini database system (MDBS). In MDBS, we will refer to one of the minicomputer systems as the controller which controls the actions of the
rest of the minicomputer systems known as back-ends. The throughput of MDBS is defined as the number of user (program) requests which are executed by the system in a second. Throughputs may also be defined for various request classes by a straightforward extension of the definition of throughput. The response time of a request in MDBS is the time between the initial issuance of the request by a user (or user program) and the final receipt of the entire response set of this request by the user (program).

1.5 Contributions of Research

In this dissertation, we present a new approach to the solution of database management problems involving database growth and performance enhancement. A system which uses a multiplicity of conventional minicomputers, novel hardware configuration and innovative software design is presented. This system tries to achieve the ideal goal of having the performance (both response time and throughput) be proportional to the multiplicity of minicomputers.

Our first effort is to identify the problems and bottlenecks involved in developing such an ideal system. Two problems, one called the controller limitation problem and the other called the channel limitation problem are
identified. Having identified these problems, our next effort is to systematically eliminate or suppress the ill-effects of these problems. We have also identified a number of other problems.

For studying the multiple back-end database system, we utilize queueing models and simulation. Queueing models and simulation are used at different design stages in order to aid the design process. Finally ours is the only comprehensive design of a multiple back-end system that covers all aspects of database management. Algorithms for the four basic request types (insert, retrieve, delete and update), algorithms for enforcing security and algorithms for enforcing concurrency control are all completely specified.

At a more detailed level, the following contributions may be cited. A solution to the task partitioning problem for multiple back-end database systems has been presented. Unlike previously proposed solutions such as the one in [Dewi78], our solution minimizes message traffic among multiple back-ends thus improving response time and throughput. Experiments have indicated that the proposed solution is extremely effective over a broad range of values for the number of back-ends. Also, for the first time, the importance of a broadcast capability in multiple back-end
database management systems has been clearly demonstrated. More importantly, the very negative impact of not having a broadcast capability has also been demonstrated. Experiments have shown us that for systems which do not have a broadcast capability, the response time will improve only with the first few back-ends. After that, the response time actually begins to increase with each additional back-end.

Eight schemes are presented for directory management in the new system. The directory management schemes are different from those for any other system. These schemes are compared in terms of system response time, throughput and storage requirements. A queueing model is used for this purpose. To the author's knowledge, this is the first time that a system has chosen a directory management policy based on analysis using queueing theory. The only other related work is that of [Chu75]. The authors of [Chu75], however, are comparing several directory management schemes for all distributed databases and not for any one specific system. Furthermore, even though the exact mathematical model which was employed in that study is not clearly described in the paper, it appears as if it was not a queueing model.

The queueing network model used in our study is a closed queueing network model. This model incorporates a separate I/O submodel for the disk subsystem. The I/O
submodel that we developed was simpler than other existing models [Bard81, Gotl73, Fran74] of disk subsystems in that we were able to obtain a closed form solution for the response time which does not rely on transformation or iteration. To a large extent, our I/O submodel is based on the work of [Gotl73]. For instance, the idea of treating the disk module queue and the channel queue as a composite queue is obtained from there. However, there are a number of differences between our work and that of [Gotl73].

Firstly, we model the channel queue differently from [Gotl73] by assuming that it is an M/G/1 queue. That is, we simplify the analysis by assuming that the requests arrive at the channel queue with exponentially distributed interarrival times. Such an assumption has been made before [Fran74]. Secondly the service time of the channel queue is always taken as the disk rotation time in our model. This is because, in our system, all requests require retrieval of entire tracks at a time. This is also different from [Gotl73]. Finally the analysis of the average seek time is also different from [Gotl73].

Our proposed I/O submodel also differs from all other models for disk subsystems in that analysis is made for bulk arrivals. Our analysis assumed bulk arrivals because of the fact that MDBS allows the user to issue requests in a
query-based language. As a result, a user query would require a number of records (and, hence, a number of disk tracks) to be accessed at the same time. Finally, a unique iteration technique is employed in order to incorporate this I/O submodel into the overall closed queueing network model.

The data placement policy is based on the work of [Wong71, Roth74]. However, it differs from all these methods in many important respects which will be discussed in Chapter IV. Very briefly, it differs from their work because we are trying to minimize response time in a multiple back-end system and they were interested only in single computer systems. Experiments to demonstrate the effectiveness of our placement policy were conducted.

The concurrency control scheme is executed at each back-end rather than at the controller. This is only possible because of the peculiarity of our architecture. In a system like DIRECT [Dewi78], concurrency control algorithms have to be run in the controller. In such a system, the performance of concurrency control algorithms cannot be improved by employing more back-ends. The design of our system has allowed us the flexibility to improve the performance of concurrency control by utilizing more back-ends. We define, for the first time, a term called monolithic consistency to describe the kind of consistency
required in a partitioned database (just like inter- and intra-consistency are needed in a centralized database and inter-, intra- and mutual consistency are needed in a distributed database). A solution which preserves monolithic consistency is presented. Our solution is unique in a number of ways. First, it advocates the use of four lock modes, instead of the traditional two lock modes. By separating the insert and delete locks from the update locks, we achieve a greater degree of concurrency. Another contribution is the identification of permutable and compatible requests for a high-level predicate-based query language (and not for a low-level one as in [Gard77]). At a practical level, a method for enforcing locking when using a predicate-based query language is proposed. Unlike [Eswa76] which uses predicate locking, our scheme is much simpler. Unlike [Jord81], the scheme allows predicate-based updates. Unlike both [Eswa76] and [Jord81], the scheme is deadlock-free. Hence, it cannot suffer from the 'starvation problem' where transactions are rolled back infinitely and are not guaranteed to complete.

The work on access control and security is based on the work in [McCa75]. However, it differs from [McCa75] in that the method of specifying access privileges is simplified and the security atoms or clusters are formed in a different way. We are also able to enforce protection down to the
attribute level. A further extension allows for the protection of access against statistical requests. Finally, the access control mechanism protects data on the basis of the relationship of the attribute value of the data and the user of the data. The security related tables are stored in the multiple back-ends. Each back-end only needs to store a subset of the tables. As a result, the response time and throughput of the entire system is improved. One final unique feature of our security mechanism is that it is the only one where the refusal of some requested access is guaranteed to lead to a saving in terms of reduced number of accesses to secondary store.

The rest of this dissertation will be organized as follows. In Chapter II, we will make a survey of typical software-oriented multiple back-end systems in existence. We shall point out the strengths and weaknesses of these existing systems and make some recommendations for MDBS so that the weaknesses pointed out for some of the systems can be avoided. In Chapter III, we argue that the attribute-based model is the "superior" or "most appropriate" data model. Accordingly, a simple data manipulation language based on this attribute-based model is chosen and presented formally. In Chapter IV, we elaborate on the notion of a cluster which will be the basic unit for access in MDBS. Furthermore algorithms for record
retrieval, insertion, deletion and update are also presented. In Chapter V, we deal with security-related issues in the MDBS. In Chapter VI, we design algorithms for concurrency control which are deadlock-free. At this point the details of MDBS will have been completely specified. Chapter VII is then a simulation model of MDBS Results showing the expected performance of MDBS as a function of the number of back-ends are presented in these sections. In Chapter VIII, we will present our final conclusions.
CHAPTER II

A SURVEY OF TYPICAL SYSTEMS AND
A STUDY OF SYSTEM ISSUES FOR DESIGN DECISIONS

In this section, we shall survey some of the existing multiple back-end systems. Their relative merits and demerits will be discussed. As we had indicated in Chapter I, our interest is in software-oriented, multiple back-end systems with a controller. We shall, therefore, neither include distributed database systems like SDD-1 [Roth80] and Distributed INGRES [Ston76a] nor hardware-oriented systems like DBC [Ban78b, Kann77b, Kann78]. Based on the findings of the survey, we shall make some design decisions for MDBS. Obviously in our design for MDBS, we will seek to avoid the weaknesses of the systems surveyed.

The systems that we shall survey in this section are RDBM [Auer80], DIRECT [Dewi78], the distributed database machine of Stonebraker [Ston78] and DBMAC [Miss80]. Even though many of these systems bear titles which include the word 'machine', they all fall into the category of
software-oriented, multiple back-end systems (with the possible exception of RDBM). These four systems do not form a comprehensive list of all systems that fall into this particular category. However, we feel that these four systems are typical of existing software-oriented, multiple back-end systems.

This survey is an important and integral part of the dissertation. We are not presenting, in this survey all the details of each of the systems surveyed. Rather, for each system surveyed, certain key observations are made. The idea is to point out the problems of these systems so that they will be used as a lesson for the better designing of MDBS.

2.1 A Survey of Typical Software-Oriented, Multiple Back-ends

2.1.1 RDBM - A Relational Database System

The RDBM consists of three major components as shown in Figure 3.

(a) A mass storage device with its own storage manager,
(b) a multiprocessor system consisting of special-function processors working on a large, common, main memory, and
RUP: Restriction and Update Processors
- - - - Data Lines
--- - Control Lines

Figure 3. RDBM - A Relational Database System
(c) a general-purpose minicomputer controlling the different hardware components and performing the preprocessing of the requests. We shall discuss each of these three major components in turn.

The mass storage device consists of conventional secondary memory, extended by a block buffer, the secondary memory manager and several processing elements, known as restriction and update processors (RUPs). A retrieve request is executed as follows in RDBM. The pages relevant to the retrieve request are identified and read from the secondary memory to the secondary memory buffer. The records from the buffer are then sent, one by one, to the next available RUP. The RUPs examine the records to determine if they satisfy the qualification criteria of the retrieve request. The RUPs forward the final set of records to the main memory. We note that, at any one time, all the RUPs are executing the same instruction (retrieve request) but on different data (records). A number of observations may be made on this architecture.
A. The Problem of Channel Limitation

First of all, advantage cannot be taken of the fact that some of the relevant pages to be searched are already in the main memory. This is because the RUPs can only examine records in the secondary memory. Secondly, it is clear that the ultimate limitation in throughput is the rate at which records can be read from the secondary memory to the RUPs via the interconnecting channel. Thus, after a certain point, the use of additional RUPs will not improve the rate at which retrieve (update, delete) requests are executed by the RDBM. We call this problem the channel limitation problem. We would prefer a system which is not limited by the speed of the channel. In other words, the throughput should not be limited by the interconnections between the secondary store and the main memory of the processors.

B. The Problem of Software Specialization

The multiprocessor component of RDBM consists of a number of special-function processors - one to perform relational joins, one to perform sorting of retrieved records, etc. In such a system, the workload distribution among the special-function processors could be uneven. For instance, if a large number of user requests require joins
to be performed, but none of these requests require any sorting to be performed, it is clear that the sort processor will be underutilized whereas the join processor is overutilized. Thus, the best utilization is not being made of these multiple processors. Furthermore, such a system may be unreliable. For example, the loss of the join processor will render the RDBM incapable of doing joins. This is known as the software specialization problem.

A system which can continue to perform all the database management functions (perhaps in a degraded mode) in spite of the loss of a processor is to be preferred to such a system where the loss of a database management function means a permanent denial of that function to the user. Therefore, to overcome this unreliable operation, a system must not have special-function software in the various processors. Rather, it should have general-purpose software in the various processors so that all of them are capable of doing all the database management functions like sorting, joining, etc. In this case, we can also expect a more even distribution of the workload among the processors. In fact, the best possible solution would be to have identical software in all the processors. Then, additional processors may be added to the system with the greatest ease because no new software has to be designed for the additional processors.
C. The Problem of Controller Limitation

The minicomputer host in RDBM controls the actions of the various hardware elements in processing a user request. Furthermore, it performs the preprocessing and analysis of the user request to determine the pages in secondary memory to be retrieved, deleted, etc. This makes the speed of the minicomputer a limiting factor to the throughput of the RDBM. To explain this point, consider, for simplicity, that all user requests require 10 seconds of minicomputer CPU time irrespective of the number of back-ends. This means that RDBM cannot support a throughput rate which is greater than six requests per minute, irrespective of how many processors it uses to speed up operations like join and sorting. We shall, henceforth, refer to this problem as the controller limitation problem.

This problem may also be explained from the viewpoint of response time. The ideal goal is to achieve a system where the response time improves in proportion to the number of back-ends used in the system. Here, the response time may be considered as the sum of the controller execution time and the back-end execution time. Addition of more back-ends can reduce the back-end execution time, but it cannot reduce the controller execution time which is a constant independent of the number of back-ends. Thus, the
controller execution time must be kept to a minimum if we are to achieve our ideal goal. This leads us to the conclusion that all major tasks must be performed in the back-end processors in a parallel fashion and that the controller must perform minimal work. So, we would like the preprocessing of user requests to be performed by the multiple back-ends in such a way that if there are n back-ends, the total time for preprocessing is speeded up by a factor of n.

D. The Problem of Data Model Limitation

RDBM supports the relational model of data. Thus, in order to support other data models such as the hierarchical or the network model, it will be necessary to convert the hierarchical or network database to a relational one. Furthermore, requests issued in a hierarchical or network data manipulation language must be translated to requests in a relational data manipulation language. Some researchers have solved the translation problem partially by translating a subset of the network model into the relational model [Katz80]. However, solutions to the problem of translating the entire network model into the relational model are not at hand. Thus, the fact that RDBM only supports the relational model is a limitation of RDBM. This is the data model limitation problem. We would prefer to have a data
model that is canonical. By a canonical data model, we mean that the model allows the entirety of any prevailing data model (i.e. relational, hierarchical and network) to be translated into the data model.

2.1.2 DIRECT - A Multiple Back-end Relational System

As a relational system depicted in Figure 4, DIRECT [Dewi78] consists of four main components: a controller, a set of query processors, a set of CCD memory modules and an interconnection matrix between the set of query processors and the set of memory modules. When the controller receives a user request, it will determine the number of query processors which should be assigned to execute the request. If the relations referenced by the request are not in the CCD memories, the controller will page them in before distributing the request to each of the query processors selected for its execution.

A retrieve request is executed as follows. First, the controller determines the optimal number of query processors that must be utilized to execute the request. This depends upon the size of the relations involved in the request, the priority of the request and the number of currently available processors. The controller then sends the request
Figure 4. The DIRECT System
to each selected processor along with information about the relations to be referenced. The controller also creates a task which waits for a done signal from each query processor. When all query processors have signalled done, the waiting task will transmit the results of the request to the user. For example, let us assume that only a single relation was referenced in the retrieve request. Then, each processor will request the controller for a page of this relation by executing a primitive called NEXTPAGE. The controller will access the secondary storage to retrieve the requested page and place it in CCD memory if it is not already there. Then, it will pass the CCD memory address of the requested page to the processor. The query processor may now access this page using the interconnection matrix and then perform the necessary work on this page. Finally, the query processor creates a temporary relation containing the selected tuples. This temporary relation will eventually be returned to the user.

A. The Problems of Hardware Specialization

We see that DIRECT overcomes the channel limitation problem by use of the interconnection matrix. Strictly speaking, this limitation will be overcome only if each query processor can access any part of the secondary memory. This is not possible in DIRECT since, only the controller
can access the secondary disk memory. The use of the CCD memories as a large cache for the secondary disk memory alleviates the channel limitation problem to a large extent. Ironically, the biggest drawback of DIRECT is also its need to use an interconnection matrix whose cost increases in proportion to the product of the number of query processors and the number of CCD page frames. While the switching delays of this interconnection matrix do not significantly affect the time to access a page of CCD memory for small number of processors and page frames, such switching delays may become significant in a full-scale system with many query processors and page frames. Another problem with the interconnection matrix is that it is not easily extensible. For example, the addition of a new CCD page frame will require modifications to the selector interfaces at each query processor, whereas the addition of a new query processor will require modifications to the interfaces at each CCD page frame. Finally the interconnection matrix is not a conventional hardware element because it must be specially designed. This is the hardware specialization problem.
B. The Problems of Control Message Traffic and Controller Limitation

Each time a new page is needed by a query processor, a message must be sent to the controller and a message must be received from the controller. These two messages may be considered as overhead for the task of reading a page from CCD memory. It has been estimated that about 8000 instructions are needed [Bora81] to send a message from the controller to a query processor or vice versa. Thus, approximately 16000 instructions have to be executed before a page may be read from CCD memory. Assuming that an instruction takes 1 usec to execute, 16 msecs of overhead are associated with the task of reading a page from CCD memory. The task of reading a page from CCD memory only takes 12 msecs [Dewi78]. Thus, the overhead associated with the task of reading a page from secondary memory is 16/28 or 57%. The above calculation does not include the time taken by the controller to search the relation tables [Dewi78] to determine the next page nor does it include the queueing delays suffered by the two overhead messages. Hence, the overhead for the task of reading a page from CCD memory is likely to be greater than the calculated figure of 57%.
Also, the present configuration of DIRECT does not permit broadcasting of requests to the query processors from the controller. As a result, a request which is to be executed by, say, three query processors would require three separate messages to be sent from the host to the query processors and this would require approximately \((8000 \times 3^e\)) 24,000 instructions and take up about 24 msecs of controller time. Thus, there is the problem of control message traffic.

Finally, we point out that DIRECT most definitely suffers from the controller limitation problem. That is, the controller is actively involved in many phases of the execution of a request - in the query analysis phase, in the concurrency control operations, in the NE*TPAGE operations, etc. - so that the throughput of DIRECT will be limited by the speed of the controller in the execution of its various tasks.

C. The Problem of Multiple Request Execution

The DIRECT approach does not allow a query processor to support concurrent execution of multiple requests. For instance, consider that while executing a retrieve request, a query processor requests a page which is not in CCD memory. Then, the query processor is idle until the page is
loaded into a page frame in CCD memory. Such idling could have been avoided if the query processors had been allowed to concurrently execute multiple requests. If each query processor were allowed to concurrently execute multiple requests, more complex software would be required in the query processors. This is the argument used by the designers of DIRECT for not allowing concurrent execution of multiple requests in the query processors. While we understand the rationale behind this argument, we still feel that there is no alternative to allowing concurrent execution in the query processors. When we have a multi-user system, the response time is of the utmost importance. By allowing concurrent execution in the query processors, we can improve the response time. Furthermore, a large fraction of the requests is likely to be I/O bound, and the use of concurrent request execution will serve to increase the utilization of the query processors (i.e., the back-ends). Even if much of this increased utilization is spent in overhead activities like task switching, concurrent request execution is still likely to provide a performance improvement.

In a system like DIRECT, concurrency control is necessary in spite of the fact that each query processor does not support concurrent request execution. This is because all the query processors are allowed to access all
the pages of CCD memory. Hence, two query processors may try to update the same page unless concurrency control is enforced. In MDBS, it will be shown that concurrency control is unnecessary if the back-ends do not support concurrent request execution. Concurrency control in DIRECT is maintained by means of lock tables residing in the controller and requires a number of messages to be exchanged between the controller and the query processors. Furthermore, locking is done at the granularity of a relation and this may reduce the degree of concurrency achievable in DIRECT. As long as no indices are maintained and each request requires the retrieval of entire relations (this is the case for DIRECT), it is difficult to have a finer locking granularity. This is because the two-phase lock protocol [Eswh76] requires that no lock be released until the end of a transaction. Thus, the lock on the first page of a relation cannot be released until the lock on the last page of that relation is set. Thus, a non-two-phase lock protocol may have to be used if a better locking granularity is to be achieved.

Another drawback of DIRECT is that after a query processor completes executing a request, it cannot immediately start executing the next request. This is because each processor does not maintain a queue of waiting requests. Only the controller maintains such a queue.
Consequently, a query processor must first wait until the results of the previous request are received at the controller and the controller has shipped the next request over to the query processor. Two messages requiring 16 msecs are needed between the end of execution of one request and the start of execution of the next. We would prefer to have a queue of waiting requests at each back-end. While such a strategy is not expected to increase throughput or decrease response time dramatically, it will certainly be an improvement over the present strategy of DIRECT. Besides the savings of sixteen msecs we had mentioned, the overhead of maintaining queues has been removed from the controller and passed on to the back-ends. This should contribute to an alleviation of the controller limitation problem. The aforementioned difficulties characterize the problem of multiple request execution.

D. The Problem of Data Model Limitation

Another criticism of DIRECT is the fact that it supports the relational model of data. As has been previously pointed out, more research needs to be done before hierarchical and network models may be supported by such a system.
In DIRECT, entire relations must be retrieved in order to answer queries. On the other hand, a system which uses index information (like inverted lists on selected attributes) will be able to retrieve relevant portions of relations and save valuable secondary storage access time.

2.1.3 The Distributed Data Base Machine of Stonebraker

A schematic of this system is shown in Figure 5. It consists of a single controller and multiple back-ends. Each back-end is connected to a single disk drive [Ston78]. The controller preprocesses the user queries and performs parsing and decomposition of user requests into requests that access only a single relation. Directory information is stored in one of the back-ends which is designated as the special back-end. After decomposing a query, the controller accesses the directory in this special back-end to determine the back-ends which must be utilized to execute the request and then sends the request over to these back-ends. After the back-ends return the results of the query to the controller, the controller outputs the results to the user that issued the query.
Figure 5. Stonebraker's Machine - A Distributed Database System
A. The Problem of Back-end Limitation

The first thing we note about the distributed database machine is that the channel limitation problem does not exist. This is because the multiple back-ends may read data from the secondary storage simultaneously via the multiple channels. However, unlike DIRECT, a request must be executed by one or more specific back-ends. For instance, if a request requires retrieval of information stored in the disk drive attached to the first back-end, only this back-end may be employed in order to execute this request. Hence, consider the following situation. Two requests are issued one after another and both require access to a relation stored entirely at the disk drive attached to the first back-end. Then, the second of these requests must wait until the first request completes execution even though many of the other back-ends in the system are idle. This characterizes the back-end limitation problem.

B. The Problem of the Specialized Back-end

The placement of the directory at a specific back-end seems quite unwarranted. Such a scheme requires that, for every request, messages must be sent to and received from this special back-end which carries the directory. This characterizes the problem of the specialized back-end.
C. The Problem of Controller Limitation

It should also be pointed out that this system suffers from the controller limitation problem, since, parsing and decomposition of requests is done entirely at the controller and will take an amount of time which is independent of the number of back-ends in the system.

D. The Problem of Multiple Request Execution

Another disadvantage of this system is that the back-ends do not support concurrent request execution. As a result, overlap of I/O time with processing of other requests is not possible and back-ends may be idle for large amounts of time waiting for an I/O to complete. Consequently, best use of processing power is not being made.

E. The Problem of Device Limitation

In this system, each back-end is only connected to a single disk drive. As a result, very large databases, of the magnitude of $10^{10}$ bytes, cannot be supported since that would require hundreds of back-ends and would make the system extremely expensive. It would seem more reasonable to allow multiple disk drives to be attached to each back-end. This is the device limitation problem.
F. The Problem of Control Message Traffic

The present configuration of the distributed database machine does not allow for broadcasting of requests to the back-ends. Hence, a request which requires cooperation among, say, three back-ends would require three separate messages from the host requiring \((8000\times3) = 24,000\) instructions and 24 msecs of controller's CPU time. The ability to broadcast the requests to the back-ends would save valuable CPU time.

G. The Problem of Data Model Limitation

Finally, a weakness of this system is that it is only designed to support the relational model of data. Furthermore, as in DIRECT, the entire relation has to be retrieved in order to answer an access request for a portion of a relation. As a result, a larger amount of information than is necessary is retrieved from secondary memory.

2.1.4 DBMAC - An Italian Database System

A schematic of this system is shown in Figure 6. Since information about this machine was obtained through private communication [Miss80], the details are necessarily very sketchy. The overall architecture consists of a controller
Figure 6. A View of the Italian Database System
connected to a set of back-end computers over a mass bus. The set of back-end computers is also connected to a set of secondary storage devices via another mass bus. There is no one-to-one correspondence between the back-ends and the secondary storage devices. Each system task (like the request preprocessing task, the concurrency control task, etc.) is performed by a set of modules which communicate with one another in the execution of that task. The set of modules for a system task are placed in such a way that, as far as possible, no two modules of a task are placed at the same back-end. Thus, all the system tasks are executed by the back-ends in a distributed fashion. Each back-end has a local primary memory and a shared primary memory. Communication among the back-end processors takes place by message passing over the first mass bus.

A. The Problem of Channel Limitation

The first observation that we wish to make about this system is that its throughput is limited by the speed of the mass bus which is attached to the secondary memory. This is because even though there are a number of back-ends, they cannot access different portions of the secondary memory simultaneously. In other words, the system suffers from the channel limitation problem. The throughput is also going to be limited by the speed of the mass bus connected.
to the controller. In fact, this mass bus is going to be heavily utilized, since, all the system tasks have been broken up into modules that communicate with each other via this mass bus.

B. The Problem of Software Specialization

Since each back-end contains a separate module from each system task, it is clear that the software in the different back-ends is not identical. This leads to a decrease in system reliability because the loss of one of the processors will render the system incapable of performing any database management function. Furthermore, such a system is not easily extensible. The addition of a new back-end will require the redistribution of the modules across the back-ends which may be a time-consuming and non-trivial task.

C. The Problem of Back-end Limitation

Unlike the distributed data base machine of Stonebraker, any of the back-ends of DBMAC may be selected to perform any of the requests, since all back-ends have access to the entire database. Consider the following example. Let us assume that each back-end has enough primary memory to store 100 tracks of information. Also,
let us assume that there are \( n \) back-ends and \( n \) disk drives and that each drive contains 1000 tracks. Then, the probability that a track needed to answer a request is in primary memory is \( \frac{100}{1000} n^{-1} \frac{1}{10} n \) (assuming that there is an equal probability of accessing any track). In an organization such as the distributed database machine, however, the probability that a track requested by a back-end is already in the primary memory is \( \frac{1}{10} \). This would cause the back-ends in DBMAC to make many more accesses to secondary memory than the back-ends in the distributed database machine.

D. The Problem of Data Model Limitation

As a final comment, DBMAC only supports the relational model of data.

2.2 Basic Design Considerations for MDBS

In this section, we will present the overall architecture of a multiple back-end database system known as MDBS. We will provide the arguments which lead us to this particular architecture. More specifically, we will present the step-by-step development of the architecture. Whenever it is necessary to make a design decision, we shall use
simulation studies and analytic techniques to examine the alternatives.

2.2.1 Nine Design Goals

In terms of our survey of typical database systems presented in the previous section, we set nine design goals for MDBS. First of all, MDBS must be designed in such a way as to avoid the channel limitation problem. Second, the controller limitation problem must be alleviated to as large an extent as possible. Third, the back-ends must execute identical software. That is, all of them must be utilized to perform all the database management functions. This leads to increased reliability and to a better workload distribution as has already been discussed. It also leads to the simplified addition of more back-ends. Fourth, communication among the back-ends and between the back-end and the controller must be kept to a minimum. Without excessive communications, the throughput of MDBS will not taper off after the first few additional back-ends. Consequently, the problem of control message traffic will not exist. Fifth, we resolve not to use any special-purpose hardware in MDBS. As a result, the problem of hardware specialization will not exist. Sixth, we propose to support concurrent request execution in our back-ends in order to
eliminate the problem of multiple request execution. For our seventh goal, we resolve to overcome the device limitation problem by attaching more than one disk drive per back-end. Eighth, we will design MDBS in such a way that all the back-ends will participate in the execution of a request. As a result, we will have eliminated the back-end limitation problem. Finally, for our ninth goal, we resolve to overcome the problem of data model limitation. In other words, we will propose a canonical data model into which all prevailing data models (such as relational, hierarchical and network) can be fully translated. If these nine design goals are attained, we believe that MDBS may come close to being an ideal system whose performance (i.e. throughput and response time) and growth are proportional to the number of back-ends employed.

2.2.2 Towards an Ideal System Architecture

In the following section, we will show how we prevent the channel limitation problem from occurring in MDBS. The proposed solution utilizes the technique used in the distributed data base machine of Stonebraker. By superimposing a data placement strategy on top of that technique, we eliminate the back-end limitation problem in MDBS as well. This strategy is explained in Section 2.3,
and further expounded in Chapter IV. Furthermore, the data placement strategy combined with the use of a broadcast capability is shown to prevent the occurrence of the control message traffic problem in MDBS. Next, the device limitation problem is eliminated in MDBS by attaching multiple disk drives to each back end. In the final section of this chapter, we will show that the problem of hardware specialization also does not exist in MDBS. Thus, five of the nine design goals are achieved in this chapter.

In Chapter III, we propose a canonical data model into which all prevailing data models and their data manipulation languages (such as relational, hierarchical and network) can be translated. Thus we eliminate the data model limitation problem in MDBS. The software to be executed at each back-end is described in Chapter IV. From the discussion in Chapter IV, it will be seen that the problems of software specialization and back-end specialization can be eliminated from MDBS and the goal of using identical software can be achieved. In order to alleviate the controller limitation problem, the directory management, security enforcement and concurrency control algorithms are carefully designed. How the careful design of these three algorithms serves to alleviate the controller limitation problem is explained in Chapters IV, V and VI, respectively. The discussion of Chapter V will center on how we choose to eliminate the
problem of multiple request execution in MDBS.

The remainder of this chapter is organized as follows. In Section 2.3, we present our data placement strategy in some detail. In Section 2.4, we will design a simulation experiment which will illustrate the importance of having a broadcast capability in MDBS. Finally, an overview of the basic design and architecture of MDBS is presented in Section 2.5.

2.3 First Design Decision - Eliminating the Channel, Back-end and Device Limitation Problems

The only architectural decision we have made with regard to MDBS up to this point is that it consists of a controller and a number of back-ends. More decisions regarding the MDBS architecture will be made as we proceed.

Let us now try to design MDBS in such a way as to eliminate any occurrence of the channel limitation problem. It was shown that RDBM and DBMAC both suffered from this problem, whereas, the distributed data base machine of Stonebraker and DIRECT did not. DIRECT overcame the problem by use of an interconnection matrix which allowed any query processor to access any CCD page frame. The distributed data base machine of Stonebraker overcame the problem by
using a separate disk drive associated with each back-end. Thus, the technique developed for DIRECT and the one developed for the distributed data base machine are good candidates for our consideration.

The technique developed for the distributed data base machine may be attractive owing to its extreme simplicity and low cost. Another strong point of this technique is that the concurrency control problem is alleviated because two different back-ends will not have the same data item for update due primarily to the use of dedicated disk drives. In spite of these advantages, the technique may be unattractive to us because it suffers from the problem that a given request can only be executed at the back-end attached to the disk drive on which the data relevant to the request resides. If the data relevant to a request resides on a single disk drive, only a single back-end can be used to execute that request. In other words, the parallelism that is present in the system is not being utilized to execute the request. We would prefer a technique which allows all the back-ends to participate in the execution of a given request.

DIRECT is a system which allows all the back-ends to participate in the execution of a request. They achieve this by bringing the data relevant to the request into a CCD
memory which is accessible from all back-ends. However, such a technique is expensive, since it requires an interconnection matrix whose cost grows as the product of the number of back-ends and the number of CCD memory modules.

We wish to develop a technique which allows all the back-ends to participate in the execution of a request on the one hand and foregoes the costly interconnection matrix on the other hand. Our solution is to have a system with dedicated secondary memories and to store the data in such a way that, whatever the request is, the data to be retrieved for satisfying the request is evenly distributed among the back-ends. Such a data placement strategy allows all the back-ends to participate in the execution of a request by reading data from the secondary memory simultaneously via the multiple channels. Thus, the lack of parallelism due to dedicated devices as exemplified in the distributed data base machine of Stonebraker does not occur in MDBS. In other words, we have taken the technique of the distributed data base machine of Stonebraker for overcoming the channel limitation problem and superimposed on it a placement strategy which serves to avoid the back-end limitation problem. An overview of MDBS architecture is depicted in Figure 7. Note that each back-end is attached to multiple disk drives for the elimination of the device limitation
Figure 7. An Overview of MDBS Architecture
2.3.1 The Need for a Data Placement Strategy

Let us illustrate the strategy first with an example. Consider a file with six records as shown in Figure 8. In the figure, an MDBS with one controller and two back-ends is depicted. In order to keep the example deliberately simple, we make the following simplifying assumptions:

1. Each back-end has only a single disk drive.
2. Each disk has only three tracks.
3. All the records are of fixed-length and occupy exactly one track.
4. Each record contains exactly three keywords.

Consider the arbitrary placement of records in Figure 8, and assume that MDBS receives the following retrieve request:

"Retrieve all records which satisfy (K1&K3)".

The query will be forwarded by the controller to the two back-ends. Given the arbitrary record placement of Figure 8, we see that the first back-end must access two tracks, since its disk contains two records - one with keywords K1, K2 and K3, and one with keywords K1, K3 and K4 - which
Figure 8. Data Placement For 6 Records
satisfy the conjunction \((K_1 \& K_3)\). The second back-end, on the other hand, does not need to access its secondary memory at all, since, it contains no records which will satisfy the given query conjunction. Thus, if we let \(t_d\) designate the time to access and read-out a track (i.e., seek time + rotation time) and we ignore the directory search time and the time taken by the controller to broadcast the request, then the response time of the request \(t_q\) is equal to \(2t_d\). Can we do better than this?

In the previous example, the poor response time was caused by an uneven distribution of the data among the two back-ends. One improved distribution of data which leads to a better response time is also shown in Figure 8. The query response time, in this case, is equal to \(t_d\). This is because each back-end has only one record satisfying the given query conjunction and needs to access only one track. Also, the two back-ends can access their respective disks simultaneously. It is this type of parallel operation, combined with the even workload (i.e. data) distribution that contributes to the optimal response time \(t_d\).

We would like to note, however, that the improved arrangement of records as shown in Figure 8 may cause the response time of some other query (e.g., "retrieve all records which satisfy the conjunction \((K_1 \& K_5)\)") to become
worse. Thus, the example has demonstrated two things to us. First, a careful placement strategy must be adopted to obtain improved response time over that which would be obtained with an arbitrary data placement strategy. Secondly, the strategy must be good for all the types of requests which will be issued against the database.

2.3.2 An Evaluation of Data Placement Strategies

Three data placement strategies are outlined and evaluated in this section. These are the exact division strategy, the track splitting with placement from the first back-end strategy, and the track splitting with random placement strategy. The differences among these three placement strategies, referred to simply as Strategy A, Strategy B and Strategy C respectively, are shown by means of an example developed in Figures 9. In Strategy A, the records in the response set of a request are divided exactly among the back-ends. Thus, if the response set of a request consists of five tracks of data and there are three back-ends as depicted in Figure 9a, each back-end will contain \((5/3)\) 1.67 tracks of data. Strategy B also consists of dividing the records of a response set equally among the back-ends. However, it is different from Strategy A in that the division of the response set takes place at
Figure 9a. Strategy A

Figure 9b. Strategy B

Figure 9c. Strategy C

Figure 9. Three Different Data Placement Strategies
track boundaries. Thus, if the response set of a request consists of five tracks of records and there are three back-ends, each back-end will contain one track of data. The remaining two tracks are then assigned to disk drives of the first two back-ends. In other words, the disk drive of the first back-end contains two tracks of the response set, the disk drive of the second back-end contains two tracks of the response set and the disk drive of the third back-end contains one track of the response set as illustrated in Figure 9b. Strategy C is very similar to Strategy B in that the division of the response set takes place at track boundaries. It differs from Strategy B in that the left-over data after exact division of the data in terms of tracks among the back-ends are assigned to the disk drives of arbitrary back-ends. Thus, in the example where the response set of a request consists of five tracks of records and there are three back-ends, the disk drive of each back-end will initially contain one track of data. The remaining two tracks are then assigned to the disk drives of two back-ends picked randomly and not necessarily to the disk drives of the first two back-ends as in Strategy B. The situation is shown in Figure 9c.

Four simulation models of MDBS are developed using SIMULA on a DEC System 20. First, we test out the simulation model in which there is no data placement policy.
That is, no assumption is made regarding where the response set of a request is located. One or more back-ends may be employed in the execution of a request depending upon where the response set to the request is stored. Parallelism will be utilized only if more than one back-end needs to be employed. Such a simulation model would approximate a system like the distributed database machine of Stonebraker [Ston78], where no special placement policy is employed. In the simulation model, a random number generator is used to determine how many back-ends will participate in answering the request and the request is then sent to these many back-ends. The remaining three simulation models simulate MDBS under the three aforementioned placement strategies.

For all four simulation models, each back-end has a queue of requests which are executed in a first-in-first-out basis. Also, the time to broadcast a request and the time to return the results to the controller are ignored since we are not interested in modelling message traffic at this point (that will be done later). Furthermore, at a back-end, the time taken to retrieve records from the secondary memory is assumed to be dominating the execution time of a request. Thus, CPU execution time is ignored. Finally, in all four models we make the assumption that a request is never satisfied by data that is already in the memory buffers of the back-end. The request always requires
data to be retrieved from the secondary memory. While such an assumption implies a worst case situation, it has the advantage that factors such as good buffering techniques will not affect our results.

The results of our simulation studies are tabulated in Tables 1 and 2. It is assumed that 25% of the requests generated are 'insert record' requests and the remaining are delete, update and retrieve requests. Each of the three latter types of requests require retrieval of information from disks. It is also assumed that anywhere between five and twenty tracks of information will have to be retrieved and searched by all the back-ends in order to answer a retrieve, delete or update request. If more than one track has to be retrieved by a back-end, no assumption that these tracks are sequentially next to each other in secondary memory is made. For example, consider an MDBS system with three back-ends and assume that the response set of a request consists of six tracks of data. Then, the data placement strategy will ensure that each back-end stores two of the six tracks of the response set. However, the two tracks of the response set that are placed at a back-end are assumed to be randomly stored on its disks. In Section 2.3.4, we will present another simulation study in which we will assume that if more than one track has to be retrieved by a back-end, then these tracks are sequentially placed on
<table>
<thead>
<tr>
<th>Inter-Arrival time of Requests (msecs)</th>
<th>Number of Back-ends = 15</th>
<th></th>
<th>Number of Back-ends = 10</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Strategy</td>
<td>Strategy A</td>
<td>Strategy B</td>
<td>Strategy C</td>
</tr>
<tr>
<td></td>
<td>No Placement Strategy</td>
<td>Strategy A</td>
<td>Strategy B</td>
<td>Strategy C</td>
</tr>
<tr>
<td>100</td>
<td>255</td>
<td>75.1</td>
<td>64.8</td>
<td>38.2</td>
</tr>
<tr>
<td>200</td>
<td>208</td>
<td>57.4</td>
<td>52.6</td>
<td>33.1</td>
</tr>
<tr>
<td>100</td>
<td>371</td>
<td>119</td>
<td>94.9</td>
<td>65.0</td>
</tr>
<tr>
<td>200</td>
<td>269</td>
<td>78.7</td>
<td>69.3</td>
<td>51.6</td>
</tr>
</tbody>
</table>

Table 1. The Response Time (in msecs) of MDBS Under Various Data Placement Strategies
The ratio = \frac{\text{Response time with no placement strategy}}{\text{Response time with best placement strategy}}

<table>
<thead>
<tr>
<th>N</th>
<th>I</th>
<th>10</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>10</td>
<td>5.71</td>
<td>6.67</td>
</tr>
<tr>
<td>200</td>
<td>10</td>
<td>5.21</td>
<td>6.28</td>
</tr>
</tbody>
</table>

N: Number of Back-end
I: Inter Arrival Time in Milliseconds

Table 2. The Improvement Caused by a Good Placement Strategy
the disk tracks of that back-end. Finally, the requests are assumed to arrive in a Poisson stream. This assumption implies that each request is independent of all others. We have also simulated the systems assuming different arrival patterns, and the results do not differ significantly from the ones for Poisson arrivals. Similarly, simulations have been run assuming a different percentage of insert requests. Once again, they do not seem to affect the results significantly and their results are not presented herein. As a final note, the method of subruns [Fran77] was used to make sure that the results were unbiased— that is, to take care of correlated observations in the simulation.

The actual response times of MDBS under the various placement strategies are shown in Table 1 for various request interarrival times and for various number of back-ends. The first observation we make from the results is that the performance of MDBS can be improved by use of a placement strategy. Among the various placement strategies, Strategy C is the best and Strategy A is the worst and this may be explained intuitively as follows. Consider a system with three back-ends and consider that a particular request's response set consists of 195 records. Furthermore, let us assume that 32 records will fit into a single track. In all the three strategies, the disk drive of each back-end would have stored two tracks of records
from the response set of this request, making a total of 192 stored records. The remaining three records would have been stored in the disk drives of the three back-ends on the basis of the placement strategy used. Strategy A ensures that the disk drives of each of the back-ends will contain exactly one of these three left-over records. Strategies B and C on the other hand, ensure that all the three left-over records are placed in a single track of the disk drive of one of the back-ends. Strategy A is no better than Strategies B and C because the time to retrieve one record from secondary memory is almost the same as the time to retrieve three records from secondary memory. This is because the minimal disk access time is the time to access a track. Let us denote the time to retrieve an entire track of records from the secondary memory as $x$. Then, in Strategy A, each back-end will spend $3x$ time units for this request. On the other hand, in Strategies B and C, only one of the back-ends will spend $3x$ time units for this request. The other two back-ends will spend only $2x$ time units for this request. This is the reason for the improved performance of Strategies B and C.

The reason why Strategy B performs worse than Strategy C may be explained as follows. In Strategy B, after dividing the tracks of a response set equally among the back-ends, the extra, say, 1 tracks are assigned to the
first 1 back-ends. As a result, back-end #1 will always take the longest time to answer a retrieve, delete or update request. The response time for this request is equivalent to the time taken by back-end #1 to answer the request, since it always takes the longest time. In general, back-end #1 will do more work and take more time than any of the other back-ends, back-end #2 will do more work and take more time than back-ends #3, #4, #5, and so on. In Strategy C, however, the workload distribution is more even owing to the fact that after initial distribution, the excess tracks are assigned randomly to the back-ends. Hence, no one back-end does more work or takes more time than any other.

In order to emphasize the advantages of a good placement strategy, i.e., Strategy C over one where no strategy is used, in Table 2, we tabulate the response time ratios in these two cases (the ratios are calculated from data in Table 1). It is seen that the response time can be improved by a factor of as much as 6.67 with good data placement. Furthermore, it is seen that for larger number of back-ends the ratio is larger. This is an interesting result, since it tells us that the affect of our proposed placement strategy will become more evident at larger number of back-ends. As we are trying to develop an extensible system, such a result is encouraging. It implies two things: (1) The response time of MDBS will be better than
the response time of a system that does not use a 'good' data placement strategy. (2) The response time of MDBS will improve as the number of back-ends is increased. The greater the number of back-ends, the better the response time.

2.3.3 An Evaluation of the Data Placement Strategies Using More Refined Assumptions

In the previous simulation study, we assumed that the tracks constituting the response set of a request were evenly distributed across all the back-ends. We also assumed that the data of the response set at any one back-end were randomly placed and not necessarily next to each other on the disk. We now focus on more refined simulation by making the assumptions more realistic. Since a large amount of data being read and manipulated by a back-end is likely stored on the disk sequentially, we will not assume that they are placed randomly on the disk. Instead, we assume now that they are likely placed sequentially, i.e., one track followed by another track. We also used more refined blocking factors. Instead of dividing data into tracks, we also consider that data will be divided into smaller units known as pages. Let there be n back-ends in MDBS. Also, let a request require, on the
average, the retrieval of $x$ records. Finally, assume that these $x$ records are stored as $s$ groups of $x/s$ records each. The value of $s$ determines the amount of sequentiality that is present in the data being retrieved. For example, in the extreme case, $s$ is of value one. This implies that all the data being retrieved is stored sequentially, since they are all in one group. The larger the value of $s$, the greater the randomness with which the records being retrieved are scattered over the disk tracks at a back-end.

For data placement, the following three new strategies are considered.

(1) Place $x/sn$ records from each group at every back-end. That is, $u(x/sn)$ records of a group are placed in some of the back-ends and $l(x/sn)$ records are placed in the remaining back-ends. Here, $u(a)$ stands for the nearest integer equal to or greater than $a$, and $l(a)$ stands for the nearest integer equal to or less than $a$.

(2) Let a page accommodate $p$ records. Then, the $x/s$ records of a group are stored in $u(x/sp)$ pages and each back-end receives $u(x/sp)/n$ pages. That is, some back-ends receive $u(u(x/sp)/n)$ pages and other back-ends receive $l(u(x/sp)/n)$ pages.

(3) Let a track store $t$ records. Then, the $x/s$
records of a group are stored in $u(x/st)$ tracks, and each back-end receives $u(x/st)/n$ tracks. That is, some back-ends receive $u(u(x/st)/n)$ tracks and others receive $1(u(x/st)/n)$ tracks.

After initial placement of data, the remaining data will be placed in the following way. For Strategy 2, some $i$ out of $n$ back-ends receive $u(u(x/sp)/n)$ pages and the remaining $n-i$ back-ends receive one page less. The $i$ back-ends which receive one page more may be the first $i$ back-ends or any $i$ consecutive back-ends starting from a randomly chosen back-end, giving rise to two variations of Strategy 2. Clearly, some other variations of Strategy 2 are also possible. However, we do not consider the other possible variations and leave them for future research. Similarly, two corresponding variations of Strategy 3 are also considered.

Strategy 1 is the same as Strategy A discussed in Section 2.3.2. Also, the two variations of Strategy 3 are similar to Strategies B and C of Section 2.3.2. The two variations of Strategy 2 are the counterparts of Strategies B and C of Section 2.3.2 where the division is done at page boundaries rather than at track boundaries.
Five separate simulation models of MDBS one for each of the aforementioned data placement strategies, are designed and evaluated using SIMULA on a DEC System 20. The assumptions made in these simulations are similar to the ones made in the simulations of Section 2.3.2 and will not be repeated here.

2.3.3.1 The Choice of a Superior Strategy for Data Placement on the Basis of Better Response Time

In discussing the new simulation results, we shall refer to Strategy 1 as the exact division strategy and to Strategy 2 as the page splitting strategy. The two variations of Strategy 2 are called page splitting with placement from the first back-end and page splitting with random placement, respectively. Similarly, the two variations of Strategy 3 are referred to as track splitting with placement from the first back-end and track splitting with random placement. The number of records needed to be retrieved for a request depends upon whether the size of a request is small or large. A small-sized request is one which requires the retrieval of a small number of records (between 1 and 320). A medium-sized request is chosen as one which requires the retrieval of between 320 and 1280 records. Large-sized requests require the retrieval of
between 1280 and 6400 records. We note that 64 records can fit in a track. Thus, the retrieval of between 320 and 1280 records is equivalent to retrieving between five and twenty tracks of records.

The number of groups into which the retrieved records fall is chosen from the set \([1, 5, 20]\). Choosing the number of groups as 1 implies that all the data is sequentially related. Thus, they form a single group. Choosing the number of groups as 20 implies on the other hand that there are 20 random data aggregates, although data in the individual aggregates may be sequentially related. Another parameter which is varied is the number of back-ends involved. This is chosen from the set \([5, 10, 15]\).

Table 3 shows the results for small-sized requests. Table 4 shows the results for medium-sized requests and Table 5 shows the results for large-sized requests.

The results indicate that the track splitting with random placement strategy is the best one over all possible values of number of back-ends, number of groups and request size. Only when the number of groups is one, is the superiority of this strategy unclear. Setting the number of groups equal to one implies that all the data being retrieved is stored sequentially. This is a rare occurrence in database systems. Should this happen, all the strategies
Number of records retrieved is between 1 and 320
Inter-arrival time = 200 msecs
Small-Sized requests
Response time results in msecs

**NUMBER OF GROUPS = 1**

<table>
<thead>
<tr>
<th>n</th>
<th>Exact Division</th>
<th>Page Splitting Round Robin</th>
<th>Page Splitting Random</th>
<th>Track Splitting Round Robin</th>
<th>Track Splitting Random</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>60.4</td>
<td>40.1</td>
<td>40.9</td>
<td>39.4</td>
<td>40.3</td>
</tr>
<tr>
<td>10</td>
<td>36.6</td>
<td>36.3</td>
<td>36.9</td>
<td>38.9</td>
<td>38.7</td>
</tr>
<tr>
<td>15</td>
<td>35.3</td>
<td>35.1</td>
<td>36.1</td>
<td>38.8</td>
<td>37.8</td>
</tr>
</tbody>
</table>

**Inter-arrival time = 200 msecs**

**NUMBER OF GROUPS = 5**

<table>
<thead>
<tr>
<th>n</th>
<th>Exact Division</th>
<th>Page Splitting Round Robin</th>
<th>Page Splitting Random</th>
<th>Track Splitting Round Robin</th>
<th>Track Splitting Random</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>358</td>
<td>343</td>
<td>343</td>
<td>501</td>
<td>89.8</td>
</tr>
<tr>
<td>10</td>
<td>338</td>
<td>317</td>
<td>298</td>
<td>485</td>
<td>66.8</td>
</tr>
<tr>
<td>15</td>
<td>332</td>
<td>306</td>
<td>252</td>
<td>485</td>
<td>57.6</td>
</tr>
</tbody>
</table>

**Inter-arrival time = 1000 msecs**

**NUMBER OF GROUPS = 20**

<table>
<thead>
<tr>
<th>n</th>
<th>Exact Division</th>
<th>Page Splitting Round Robin</th>
<th>Page Splitting Random</th>
<th>Track Splitting Round Robin</th>
<th>Track Splitting Random</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>977</td>
<td>929</td>
<td>726</td>
<td>1250</td>
<td>266</td>
</tr>
<tr>
<td>10</td>
<td>969</td>
<td>886</td>
<td>421</td>
<td>1230</td>
<td>159</td>
</tr>
<tr>
<td>15</td>
<td>961</td>
<td>877</td>
<td>305</td>
<td>1230</td>
<td>130</td>
</tr>
</tbody>
</table>

Table 3. Response Time Results for the Various Strategies for Small-Sized requests
Number of records retrieved is between 320 and 1280
Inter-arrival time = 200 msecs
Medium-Sized requests
Response time results in msecs

<table>
<thead>
<tr>
<th>Number of Groups = 1</th>
<th>n</th>
<th>Exact Division</th>
<th>Page Splitting Round Robbin</th>
<th>Page Splitting Random</th>
<th>Track Splitting Round Robbin</th>
<th>Track Splitting Random</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>75.9</td>
<td>75.6</td>
<td>77</td>
<td>74.8</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>52.8</td>
<td>52.6</td>
<td>53.4</td>
<td>50.6</td>
<td>51.2</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>45.8</td>
<td>45.5</td>
<td>47.4</td>
<td>44.6</td>
<td>46.9</td>
</tr>
</tbody>
</table>

Inter-arrival time = 200 msecs

<table>
<thead>
<tr>
<th>Number of Groups = 5</th>
<th>n</th>
<th>Exact Division</th>
<th>Page Splitting Round Robbin</th>
<th>Page Splitting Random</th>
<th>Track Splitting Round Robbin</th>
<th>Track Splitting Random</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>617</td>
<td>597</td>
<td>680</td>
<td>535</td>
<td>244</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>432</td>
<td>421</td>
<td>420</td>
<td>504</td>
<td>143</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>389</td>
<td>379</td>
<td>379</td>
<td>494</td>
<td>109</td>
</tr>
</tbody>
</table>

Inter-arrival time = 1000 msecs

<table>
<thead>
<tr>
<th>Number of Groups = 20</th>
<th>n</th>
<th>Exact Division</th>
<th>Page Splitting Round Robbin</th>
<th>Page Splitting Random</th>
<th>Track Splitting Round Robbin</th>
<th>Track Splitting Random</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>1060</td>
<td>1050</td>
<td>1000</td>
<td>1250</td>
<td>246</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1010</td>
<td>996</td>
<td>987</td>
<td>1230</td>
<td>159</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>996</td>
<td>969</td>
<td>868</td>
<td>1230</td>
<td>131</td>
</tr>
</tbody>
</table>

Table 4. Response Time Results for the Various Strategies for Medium-Sized Requests
Number of records retrieved is between 1280 and 6400

Inter-arrival time = 500 msecs

Large-Sized requests

Response time results in msecs

<table>
<thead>
<tr>
<th>n</th>
<th>Exact Division</th>
<th>Page Splitting Round Robin</th>
<th>Page Splitting Random</th>
<th>Track Splitting Round Robin</th>
<th>Track Splitting Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>238</td>
<td>237</td>
<td>245</td>
<td>235</td>
<td>242</td>
</tr>
<tr>
<td>10</td>
<td>120</td>
<td>119</td>
<td>121</td>
<td>119</td>
<td>121.0</td>
</tr>
<tr>
<td>15</td>
<td>87.1</td>
<td>86.9</td>
<td>91.1</td>
<td>86.2</td>
<td>92.2</td>
</tr>
</tbody>
</table>

Inter-arrival time = 500 msecs

NUMBER OF GROUPS = 5

<table>
<thead>
<tr>
<th>n</th>
<th>Exact Division</th>
<th>Page Splitting Round Robin</th>
<th>Page Splitting Random</th>
<th>Track Splitting Round Robin</th>
<th>Track Splitting Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>517</td>
<td>513</td>
<td>571</td>
<td>491</td>
<td>465</td>
</tr>
<tr>
<td>10</td>
<td>308</td>
<td>305</td>
<td>312</td>
<td>290</td>
<td>265</td>
</tr>
<tr>
<td>15</td>
<td>258</td>
<td>256</td>
<td>266</td>
<td>247</td>
<td>218</td>
</tr>
</tbody>
</table>

Inter-arrival time = 1000 msecs

NUMBER OF GROUPS = 20

<table>
<thead>
<tr>
<th>n</th>
<th>Exact Division</th>
<th>Page Splitting Round Robin</th>
<th>Page Splitting Random</th>
<th>Track Splitting Round Robin</th>
<th>Track Splitting Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1580</td>
<td>1560</td>
<td>1540</td>
<td>1360</td>
<td>866</td>
</tr>
<tr>
<td>10</td>
<td>1220</td>
<td>1210</td>
<td>1240</td>
<td>1270</td>
<td>435</td>
</tr>
<tr>
<td>15</td>
<td>1130</td>
<td>1110</td>
<td>1090</td>
<td>1250</td>
<td>307</td>
</tr>
</tbody>
</table>

Table 5. Response Time Results for the Various Strategies for Large-Sized Requests
are approximately equal in terms of the resulting response times. However, the track splitting with random placement strategy leads to very dramatic improvements in response time over the next best strategy for larger number of groups. The superiority of this strategy over all other strategies is most evident for small to medium-sized requests and when the data tends to be more randomly distributed. Situations when the average response time of MDBS using the next best strategy is five times slower than the response time of MDBS using the track splitting with random placement strategy are noticed. For instance, for small-sized requests, when the number of groups is five and the number of back-ends is fifteen, the track splitting with random placement strategy leads to an average response time of 57.6 msecs. The nearest rival, which is the page splitting with random placement strategy, leads to a response time of 252 msecs which is almost five times slower. The track splitting strategy performs better than the page splitting strategy for exactly the same reason that Strategy C of Section 2.3.2 performs better than Strategy A of the same section.

Thus, from a performance point of view, the best data placement strategy is the one where the response set is divided up into tracks and stored as multiples of data tracks. Should there be extra data tracks after even
distribution among the back-ends, they are assigned to the disks of consecutive back-ends starting from a randomly chosen back-end.

In Chapter IV, we will relate the notion of clusters with the notion of groups. Clusters are formed on the basis of the attributes and their potential utilization. However, the placement of the clusters is related to the placement strategies of the groups which relies on the present simulation studies.

2.3.3.2 The Choice of a Superior Data Placement Strategy on the Basis of Better Storage Utilization

Let us now consider the various strategies from the point of view of storage utilization. In this case, the comparison is between a strategy which stores groups of records in multiples of tracks (as in Strategy 3) and a strategy which stores groups of records in multiples of pages (as in either Strategy 1 or 2). Let us call the former the track splitting strategy and the latter the page splitting strategy.

There are two factors affecting storage utilization: First, there is the unutilized space owing to the fact that records do not exactly fit in a page (or a track). Thus, if
A page can hold only 512 bytes and each record is of size 200 bytes, 112 bytes on each page is wasted. The second factor that leads to unutilized space is the fact that each group of records ends on page boundaries (or track boundaries) and records from two different groups are never placed on the same page (or track). The first factor is favorable to track splitting strategies and the second to page splitting strategies.

Consider, once again, that there are x records to be retrieved and that they are to be retrieved as s groups of x/s records each. In the page splitting strategy, the x/s records are stored in u(x/sp) pages. The wasted space in u(x/sp)-1 of these pages is the difference between the page size and the p record sizes. The wasted space on the last page is

\[
\text{page size} - \left(\frac{x}{s} - (u(x/sp)-1)p\right) \times \text{record size}
\]

Let page size = ps, record size = rs, and track size = ts. Then, percentage of wasted space in the page splitting strategy is

\[
\frac{(u(x/sp)-1)(ps-p*rs)}{(u(x/sp)ps)} + \frac{(ps - \left(\frac{x}{s} - (u(x/sp)-1)p\right)rs)}{(u(x/sp)ps)}
\]

Reasoning similarly the percentage of wasted space in the track splitting strategy is

\[
\frac{(u(x/st)-1)(ts-t*rs)}{(u(x/st)ts)} + \frac{(ts - \left(\frac{x}{s} - (u(x/st)-1)t\right)rs)}{(u(x/st)ts)}
\]
The percentage of wasted space will depend on the size of a record and the number of records in a group. Some experimental results are shown in Table 6. The record size is chosen from the set \([200, 300, 400]\) bytes, and the value of \(x/s\) is chosen from the set \([20, 50, 100]\). The results are as follows. In either strategy, the wasted space decreases when the record size is increased. The page splitting strategy has less wasted space when the amount of sequentiality in the data being retrieved is low. However, if a good clustering policy is employed so that more of the records that are going to be retrieved together are stored together, then the track splitting strategy will waste less space. In Chapter IV, we will present such a policy. In conclusion, we shall use the track splitting strategy in MDBS both for the good response time and for the low storage wasted.

2.3.4 Next Step in the Design Process

We have now gone one step further in our design of MDBS. We have decided that it is going to be a dedicated system in which each back end has attached to it a number of disk drives which are not accessible from any other back-end. As a result, the controller and device limitation problems are eliminated in MDBS. Furthermore, we have
### PAGE-SPLITTING POLICY

<table>
<thead>
<tr>
<th>Record Size (bytes)</th>
<th>30</th>
<th>60</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>.219</td>
<td>.219</td>
<td>.219</td>
</tr>
<tr>
<td>300</td>
<td>.414</td>
<td>.414</td>
<td>.414</td>
</tr>
<tr>
<td>400</td>
<td>.219</td>
<td>.219</td>
<td>.219</td>
</tr>
<tr>
<td>500</td>
<td>.023</td>
<td>.023</td>
<td>.023</td>
</tr>
</tbody>
</table>

### TRACK-SPLITTING POLICY

<table>
<thead>
<tr>
<th>Record Size (bytes)</th>
<th>30</th>
<th>60</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>.625</td>
<td>.023</td>
<td>.023</td>
</tr>
<tr>
<td>300</td>
<td>.438</td>
<td>.023</td>
<td>.023</td>
</tr>
<tr>
<td>400</td>
<td>.250</td>
<td>.023</td>
<td>.023</td>
</tr>
<tr>
<td>500</td>
<td>.063</td>
<td>.023</td>
<td>.023</td>
</tr>
</tbody>
</table>

Table 6. Comparison of Storage Wastage Between Two Different Placement Policies are Used.
decided to use a data placement strategy in order to store the data in the disk drives in an evenly distributed manner for the improvement of the response time and storage utilization. More details of the strategy will be presented in Chapter IV. We also argue that such hardware configuration and data placement will ensure that all back-ends will participate in the execution of a request and eliminate the back-end limitation problem. Unlike DIRECT, we do not need a costly interconnection matrix in order to achieve this. Finally, we will show in the next section that as a result of our data placement strategy the amount of control message traffic needed in MDBS is much less than in DIRECT. We will also propose, in the next section, the incorporation of a broadcast capability in MDBS for further minimizing control message traffic.

24 Second Design Decision - Minimizing the Problem of Control Message Traffic

Consider a DIRECT system with three query processors executing a request that requires access to nine pages of memory. The execution of such a request requires three messages to be sent from and received by each of the three query processors. It also requires the controller to send and receive nine messages. The above figures are only for
messages that are sent and received by the processors asking for page frames and do not include the three messages that must be sent by the controller in order to initiate the query processors nor does it include one message from each of the query processors when they output the results. In all, DIRECT exchanges 24 messages while executing this request for nine pages.

MDBS, on the other hand, will require only six messages. Three messages are needed to send the request to the three back-ends and one message will be received from each of the back-ends when they output the results.

The number of messages needed in MDBS may be further reduced to four if we have a broadcast capability. Our feeling that a broadcast capability is important for MDBS is prompted by the following. Since the data placement strategy ensures that all back-ends will be participating in answering a request, the request must now be sent to every back-end. Thus, with a broadcast capability in a system with n back-ends, we need only a single message rather than n messages to broadcast a request. The simulation experiment in the next section illustrates, graphically, the advantages of a broadcast capability.
2.4.1 The Need for a Broadcast Capability

Two sets of simulations are run - one for MDBS without the broadcast capability and one for MDBS with such a capability. Once again, the simulation programs are written in SIMULA on a DEC System 20. It is assumed, as in the earlier simulation experiments, that four types of requests (i.e., retrieve, insert, delete and update) are issued and that 25% of all requests are of the insert type. Also, as before, it is assumed that anywhere between five and twenty tracks will have to be retrieved and searched in order to answer a retrieve, delete or update request. The requests are assumed to arrive in a Poisson stream. Finally it is assumed that the time taken to retrieve records from secondary memory predominates over CPU processing time so that the latter may be ignored. Simulations are run for various number of back-ends and various request inter-arrival times. The response time results are tabulated in Table 7.

Table 7 tells us that the inclusion of a broadcast capability can make the MDBS system up to 2.5 times as fast as it would be without such a capability. Also, the utility of such a capability becomes more evident for larger number of back-ends. This is, of course, to be expected. It should be pointed out that the improved response time in the
<table>
<thead>
<tr>
<th>System</th>
<th>MDBS with Broadcast</th>
<th>MDBS without Broadcast</th>
<th>MDBS without Broadcast/MDBS with Broadcast</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inter Arrival Time of Requests (msecs)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>46.9</td>
<td>104</td>
<td>2.22</td>
</tr>
<tr>
<td>200</td>
<td>40.7</td>
<td>104</td>
<td>2.56</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>System</th>
<th>MDBS with Broadcast</th>
<th>MDBS without Broadcast</th>
<th>MDBS without Broadcast/MDBS with Broadcast</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inter Arrival Time of Requests (msecs)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>72.7</td>
<td>92.6</td>
<td>1.27</td>
</tr>
<tr>
<td>200</td>
<td>59.2</td>
<td>92.2</td>
<td>1.56</td>
</tr>
</tbody>
</table>

Table 7. Response Time for MDBS with and without Broadcast Facility
case of MDBS with the broadcast capability is a direct result of this capability, since the control message traffic is reduced.

2.4.2 An Evaluation of the Broadcast Capability With More Refined Assumptions

The results of Table 7 assumed that the tracks to be retrieved from the secondary memory of a back-end were distributed randomly across the disk drives of that back-end. Thus, the time to retrieve a track includes both seek and rotation times. Let us now consider that the tracks to be retrieved at a back-end are clustered in such a way that the seek time may be unnecessary for all except the first track retrieved. The number of back-ends is chosen from the set [5, 10, 15]. Another parameter that is varied is the number of tracks that must be retrieved for a typical request. A small-sized request requires the retrieval of between one and five tracks. A medium-sized request requires the retrieval of between five and twenty tracks. Finally, for large-sized requests, the number of tracks to be retrieved varies from twenty to one hundred. The results for small-sized requests are shown in Table 8, the results for medium-sized requests are shown in Table 9, and the results for large-sized requests are shown in Table 10.
Response Time Tables (in milliseconds)
Small-Sized Requests
(Between 1 and 5 tracks)

<table>
<thead>
<tr>
<th>Number of back-ends</th>
<th>Inter-arrival time = 100 msecs</th>
<th>Inter-arrival time = 200 msecs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>With Broadcast</td>
<td>Without Broadcast</td>
</tr>
<tr>
<td>5</td>
<td>34.5</td>
<td>46.3</td>
</tr>
<tr>
<td>10</td>
<td>21.8</td>
<td>59.3</td>
</tr>
<tr>
<td>15</td>
<td>18.1</td>
<td>86.9</td>
</tr>
</tbody>
</table>

Table 8. Comparing MDBS with and without Broadcast for Small-Sized Requests
Response Time Tables (in msecs)
Medium-Sized Requests
(Between 5 and 20 tracks)

<table>
<thead>
<tr>
<th>Inter-arrival time = 100 msecs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of back-ends</td>
</tr>
<tr>
<td>----------------------</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inter-arrival time = 200 msecs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of back-ends</td>
</tr>
<tr>
<td>----------------------</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>15</td>
</tr>
</tbody>
</table>

Table 9. Comparing MDBS with and without Broadcast for Medium-Sized Requests
Response Time Tables (in msecs)
Large-Sized Requests
(Between 20 and 100 Tracks)

<table>
<thead>
<tr>
<th>Number of back-ends</th>
<th>With Broadcast</th>
<th>Without Broadcast</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>233</td>
<td>232</td>
</tr>
<tr>
<td>10</td>
<td>118</td>
<td>152</td>
</tr>
<tr>
<td>15</td>
<td>86.8</td>
<td>152</td>
</tr>
</tbody>
</table>

Table 10. Comparing MDBS with and without Broadcast for Large-Sized Requests
Consider the results for the case when the number of tracks to be retrieved varies from one to five. With an interarrival time of one request every 200 msecs, the average response times for 5, 10 and 15 back-ends using broadcast is 31.7, 20.6 and 17.2 msecs, respectively. Thus, the response time decreases with increasing number of back-ends as expected. On the other hand, the corresponding response times for the case when no broadcast capability is assumed is 46.2, 59.3 and 87.0 msecs, respectively. That is, the response time actually increases with increasing number of back-ends! This is because the message sending time is predominating over the secondary memory access time (since the number of tracks to be retrieved is so few). By comparing the set of response times with the broadcast capability with the corresponding set of response times without the broadcast capability, we see that the use of broadcast may result in improvements of 50%, 130% and 400% for the respective configurations of 5, 10 and 15 back-ends. The results are approximately the same even when the interarrival rate is increased to one request every 100 msecs.

Next, let us consider the results for the case when the requests are medium-sized and require the retrieval of between five and twenty tracks of data. Once again, the response time of MDBS without the broadcast capability
increases with increasing number of back-ends instead of decreasing. Thus, with an interarrival time of one request every 100 msecs and five back-ends, the response time is 85.1 msecs. However, the response time with ten back-ends is 87.4 msecs and with 15 back-ends it is 104 msecs. On the other hand, the system with the broadcast capability behaves in an increasingly better manner. The corresponding figures in this case are 96.4, 56.7 and 43.2. The percentage of improvement gets to be as high as 250% or 2.5 times better.

Finally consider the results for the case when the number of tracks retrieved is between twenty and one hundred. The effect of not having a broadcast capability is expected to be felt the least under such circumstances. The simulation results certainly bear out this intuitively expected result. For the first time, the system without the broadcast capability shows an improvement in response time when the number of back-ends is increased from five to ten. However, the performance of the system does not improve when the number of back-ends is increased from ten to fifteen. The performance of the system with the broadcast capability improves constantly with increasing number of back-ends. When the number of back-ends is 15, we note that the system with the broadcast capability still outperforms the system without the broadcast capability by 90%.
In conclusion, the use of a broadcast capability can be very important. The need becomes more acute when the number of back-ends is large and when typical requests do not require the retrieval of large amounts of data. A system without broadcast capability can behave anomalously in that the response time actually increases with increasing number of back-ends. Such degradation in response time is primarily due to the increase in control message traffic.

We wish to emphasize that the ability to broadcast is essentially attained by software means. That is, no special-purpose hardware is needed in order to achieve a broadcast capability. There are many examples of systems that provide a broadcast capability. For instance, Ethernet [Metc76], which is a coaxial cable network, provides such a capability. Another scheme which provides a broadcast capability is the time-shared bus. Three different commonly used techniques for achieving a broadcast capability using a time-shared bus are described in [Tane81] and will not be repeated here. In conclusion, broadcast capability is achieved by using appropriate software and requires no special-purpose hardware. Furthermore, such software is available from many vendors and is being commonly used. Hence, our proposal for using a broadcast capability in MDBS does not lead to the problem of hardware specialization.
2.5 An Overview of the MDBS Architecture and Design

At this point, our proposed architecture looks as shown in Figure 7. It consists of a single controller attached to a number of back-ends by way of a bus or a local network (such as Ethernet). A back-end is attached with a number of disk drives which may be accessed only by that back-end. Since we do not use special-purpose hardware, we have therefore solved the problem of hardware specialization in MDBS. In Chapter I, we had listed several different issues which had to be studied in the context of a multiple back-end system. Let us briefly consider those issues which have been resolved for MDBS at this point of the design study.

The first issue was regarding the optimal way of interconnecting a large number of back-ends and the optimal way of connecting the controller to the back-ends. We have decided on a scheme whereby all the back-ends and the controller are attached to a bus. This provides the controller with the ability to broadcast requests to all the back-ends and leads to a minimization of the control message traffic problem.

Another issue concerns the placement of data aggregates of the database across the back-ends. This issue has also been resolved to our satisfaction. We have shown, by
simulation, that a strategy whereby the response sets to all requests are distributed evenly among the back-ends and where the extra tracks are assigned starting from a random back-end is the best strategy both in terms of minimum response time and in terms of minimum wasted storage space. We have termed it the track splitting with random placement strategy. This strategy eliminates the back-end limitation problem.

Another issue we had mentioned in Chapter I was the database store interconnection problem - that is, the attachment of disk drives to back-ends. We have chosen to adopt the dedicated approach because it eliminates the back-end limitation problem.

The next issue concerns the execution strategy. That is, should a SIMD or a MIMD approach be used. Our proposed design operates in a MIMD fashion. As much as possible, the different back-ends are made to work on the same request. However, when a back-end finishes the execution of a request, the back-end is to start the execution of the next request. In other words, the different back-ends are executing requests in an asynchronous fashion. Asynchronous execution is achieved in MDBS by having a queue of requests at each back-end. Whenever a back-end finishes executing a request, it picks up the next request from its queue and
begins its execution. As we had indicated in our survey of DIRECT, this saves the need for a back-end to send a message to the controller after execution of each request and the need for the controller to respond with the next request in the controller queue. Thus assuming a message time of eight milliseconds, this proposal will save sixteen milliseconds per request over a proposal where no queue of requests is maintained at each back-end. Additionally it alleviates the controller limitation problem to some extent by removing the queue handling software from the controller and placing it at the back-ends. In fact, the amount of overhead associated with the controller is exactly the same as would have been associated with it if the system had been executing in an SIMD mode. In other words, our proposal for a queue at each back-end allows us to reap the benefits of the MIMD mode of execution at the price of the SIMD mode of execution.

Thus, five of the nine design goals we set for ourselves in Section 2.2.1 have been achieved at this point. We will achieve the goal of finding a canonical data model in Chapter III. We will achieve the goal of using identical software in Chapter IV. The means for achieving the goal of eliminating the multiple request execution problem is described in Chapter VI. Finally, in order to achieve our goal of alleviating the controller limitation problem, the
Directory management security enforcement and concurrency control algorithms are carefully designed. These algorithms are described in Chapters IV, V and VI, respectively.
CHAPTER III

THE CHOICE OF A DATA MODEL AND A DATA MANIPULATION LANGUAGE

In this chapter, we will develop a canonical data model for MDBS implementation. A canonical model must meet the following three criteria: the translation criterion, the partition criterion and the language criterion. These criteria will be elaborated in the following sections. An attribute-based model is proposed herein as the best data model to meet the criteria. With the canonical data model as a basis, we then propose a data manipulation language in which users may issue requests to MDBS. The language also encompasses the useful notion of a transaction.

3.1 Three Selection Criteria

We will discuss the criteria informally and briefly herein. A detailed discussion of the criteria will be included in the course of developing the canonical model,
known as the attribute-based model.

3.1.1 The Translation Criterion

All the systems that we surveyed in Chapter II [Auer80, Dewi78, Miss80, Ston78] support the relational model of data. On the other hand, a large number of operational database systems [Datayy, Idmsyy, Systyy, Adabyy, Totayy] support either the hierarchical or the network data model. Therefore, there is the need to translate the existing hierarchical and network databases into relational databases before they may be used on relational systems. Such translation is part of the general studies for converting a database from one model to another and is known as \textit{database transformation} [Bane80]. Furthermore, there is the need to translate the requests for the hierarchical or network database into requests for a relational database. Request translation is also part of general studies for translating one data language to another and is known as \textit{query translation} [Bane80]. To the best of our knowledge, complete solutions to the problems of database transformation and query translation among the aforementioned three models are not at hand. Hence, more research needs to be done before hierarchical and network databases may be supported entirely on a relational system.
For this reason, we do not prefer the relational model. Similar reasons may be raised to reject the network and hierarchical data models. Database transformation and query translation constitute, therefore, the first criterion, i.e., the translation criterion, for selecting a data model.

There are two more criteria which a data model must satisfy if we are to implement it in an MDBS. These are referred to as the partition criterion and the language criterion. We will illustrate these two criteria by means of examples in the sequel. The examples will also illustrate why we choose to reject the network and hierarchical data models for MDBS implementation.

3.1.2 The Partition Criterion

Consider the sample network data model shown in Figure 10 where an inventory control database consists of four record types (namely, customer, order, item and part) and three set types (customer-order, order-item and part-item). The customer-order set links each customer to all his orders. Thus, Customer ABC is linked to orders 1, 2 and 3. Corresponding to each order, there can be any number of items. Thus, for example, items 1 and 2 correspond to order 1. Finally, a part in the inventory consists of a number of
Figure 10. A Sample Network Database with Four Record Types and Three Set Types
items. For example, part 2 consists of items 2, 3, 5, 7 and 8.

A typical request issued to such a database may be as follows:

'list all the order numbers for Customer ABC'.

A conventional network database system would respond to such a request by first retrieving the record for Customer ABC (by using a hashing function on the unique customer number) and then following the chain of pointers to make three additional secondary accesses to retrieve the records for orders 1, 2 and 3. Thus, a conventional system would need four secondary accesses to respond to this request.

Consider, now, how a system like MDBS would answer this request. For expository purposes, let us assume that the MDBS consists of a controller and three back-ends. Now the response time of MDBS to this particular request would depend upon the manner in which the database of Figure 10 is distributed across the three back-ends. If the record corresponding to Customer ABC, and those corresponding to orders 1, 2 and 3 are all stored at one of the back-ends, then MDBS, like the conventional system, would also need four secondary memory accesses. Clearly, we can do better
than four.

One way to minimize the number of accesses might be to partition the database in such a way that the three records in the customer-order set with Customer ABC as owner are placed in the three different back-ends. The new database storage is now shown in Figure 11. Essentially, we have partitioned the member records of the set customer-order across the back-ends. Thus, any request which requires retrieval of all members in any customer-order set may be easily answered. The same request

'list all order numbers for Customer ABC'

is now answered in two accesses to the secondary memory, instead of the four needed in the previous environment.

However, we have had to incorporate some redundancy in the database. Thus, the Customer ABC record has now to be stored in three different disks, the Customer DEF record has to be stored in two disks, the Part 1 record has to be stored in two disks and the Part 2 record has to be stored in three disks. The record storage increases by a factor of \((\frac{24}{18})\) 1.5. Similarly, the number of pointers has gone up from 31 to 37.
Figure 11: Partitioning the database of Figure 10 on 3 back-ends
In addition to the fact that the total amount of storage may go up is the new problem of updating redundant data. Thus, if the address of Customer ABC is changed, it must now be changed in three different back-ends. Hence, the controller must be aware of all the different copies of a record and must ensure the mutual consistency [Thom79] of all these different copies during update. Algorithms for ensuring the mutual consistency of these different copies are rather complicated. Thus, we should avoid the problem if possible.

Although the above solution to partitioning the network database has resulted in faster response to requests for all members in any customer-order set, the response to requests for all members in an order-item set is as slow as in any conventional system. In order to improve the response time to such requests, we will need to partition the member records of all order-item sets across the back-ends. This causes a further increase in the amount of data redundancy.

Finally, we see that the insertion of a new record into the database will require the addition of other data. In referring to Figure 11, for example, if a new order is created for Customer DEF and must be inserted into the database, it should be inserted into back-end number 3 in order to ensure an even distribution of the member records.
of Customer DEF across the back-ends. However, since back-end 3 does not contain a record for Customer DEF, a copy of the Customer DEF record must be created in back-end 3 in order to represent the relationship between Customer DEF and the newly inserted order record.

From the above discussion, we learn that the partitioning of a network database for MDBS may introduce the problems of storage redundancy and mutual consistency. A data model is easily partitionable if the partitioned database stored in MDBS for that model leads to little or no storage redundancy. Consequently, there will be little or no concern for the problem of mutual consistency during update. Both hierarchical and network models are not easily partitionable. On the other hand, the relational model is easily partitionable. This completes our presentation of the partition criterion. It also illustrates why we are against implementing a network or hierarchical data model in MDBS.

3.1.3 The Language Criterion

Another reason for not implementing the hierarchical or network data model in MDBS is related to the data manipulation languages associated with these data models,
i.e., the so called language criterion. It is evident that MDBS will outperform conventional systems for requests that require content-addressing and retrieval of large volumes of data, because the data can be spread across the various back-ends and can be fetched in parallel. However, if the user transaction demands records in a sequential, one record at a time manner, then MDBS cannot outperform a conventional system to any great extent. Unfortunately the data manipulation languages associated with hierarchical and network databases tend to manipulate data in a sequential, one-record-at-a-time manner. Hence, they are unsuitable for implementation in MDBS. The ideal language for MDBS will be one which is highly concurrent and which requires the retrieval of large volumes of data. We shall present such a language in a later section.

3.2 The Attribute-Based Model

Having eliminated from consideration the relational, hierarchical and network data models, we shall now consider the attribute-based data model [Hsia70, Roth74, Wong71]. In [Bane77, Bane78a, Bane80], several contributions are made with regard to the attribute-based model. First, they have shown that any relational, network or hierarchical database may be transformed, in a straightforward way, to an
attribute-based database. Thus, there is no database transformation problem. They have also demonstrated that the requests issued in the data manipulation languages of these three data models may be easily translated into the requests of the attribute-based data manipulation language. Thus, the attribute-based data model does not suffer from the query translation problem. Consequently, unlike the relational, hierarchical and network data models, the attribute-based model meets the translation criterion. Why is it that other data models and their manipulation languages can be so easily translated into the attribute-based model and its manipulation language? The reason has to do with the fact that the attribute-based model is a very simple model which embodies only a few simple concepts. When one tries to transform the database of a complex model into a database of another complex model, one must use the complex concepts in the second model to 'emulate' the complex concepts of the first model. This is difficult due mainly to the major differences among the concepts of these models. For example, the concept of a set of the network model is sufficiently complex as to make it difficult to find its counterpart in the relational and hierarchical models. In other words it is not easy to find concepts in the relational and hierarchical models to emulate the concept of a set. On the other hand, being
basic, the concepts of the attribute-based model may be used as building blocks for the more complex concepts of the aforementioned three data models. The process of translating a complex concept into one or more elementary concepts is frequently an easier task than the process of translating a complex concept to one or more complex ones.

Next, the attribute-based data model does not suffer from the partition problem. We recall that the network data model suffers from the partition problem because its partitioned database has a large amount of data redundancy. Such redundancy was essentially caused by the fact that two different mechanisms are used to represent data in a network database, where entities are represented by records and relationships are represented by pointers. In an attribute-based model, all logical concepts (i.e., entities and relationships) are represented by attribute-value pairs. Thus, data may be easily partitioned across the various back-ends with no redundancy. Finally, the data manipulation language of the attribute-based model does not operate in a sequential, record-at-a-time manner. By the use of boolean expressions of predicates as queries, it operates in a highly concurrent manner thus allowing us to utilize the capabilities of MDBS to the fullest. Accordingly we shall choose to implement the attribute-based model, since it meets all three criteria,
namely, the translation, partition and language criteria.

3.2.1 Concepts and Terminology

The smallest unit of data in MDBS is a keyword which is an attribute-value pair, where the attribute may represent the type, quality, or characteristic of the value. Information is stored in and retrieved from MDBS in terms of records; a record is made up of a collection of keywords and a record body. The record body consists of a (possibly empty) string of characters which are not used for search purposes. For logical reasons, all the attributes in a record are required to be distinct. An example of a record is shown below:

(\langle File, EMP \rangle, \langle Job, MGR \rangle, \langle Dept, TOY \rangle, \langle Salary, 30000 \rangle).

The record consists of four keywords. The value of the attribute Dept, for instance, is TOY. In this dissertation, we will use "<", ">", "(", ")" to bracket an attribute-value pair; "(", ")" to parenthesize a record; character strings with leading capitals for attributes; numerals or all capitals for values; and commas to separate the attribute-value pairs of a record.
MDBS recognizes several kinds of keywords: simple, security and directory. Simple keywords are intended for search and retrieval purposes. Security keywords are intended for access control and will be described more fully in Chapter V. Directory keywords are used for forming clusters. A cluster of records has a high probability of being retrieved from the back-ends together. Records of a cluster are therefore stored in close proximity. We will discuss the concept of a cluster and cluster algorithms in Chapter IV.

A *keyword predicate*, or simply *predicate*, is of the form \((\text{attribute}, \text{relational operator}, \text{value})\). A *relational operator* can be one of \([=,\neq,>,\geq,<,\leq]\). A keyword \(K\) is said to *satisfy* a predicate \(T\) if the attribute of \(K\) is identical to the attribute in \(T\) and the relation specified by the relational operator of \(T\) holds between the value of \(K\) and the value in \(T\). For example, the keyword \(<\text{Salary},15000>\) satisfies the predicate \((\text{Salary}>10000)\).

A *descriptor* can be one of three types:

**Type-A:** The descriptor is a conjunction of a *less-than-or-equal-to* predicate and a *greater-than-or-equal-to* predicate, such that the same attribute appears in both predicates.
example of a type-A descriptor is as follows:

\[(\text{Salary} \geq 2,000) \& (\text{Salary} \leq 10,000)\].

More simply this is written as follows:

\[(2,000 \leq \text{Salary} \leq 10,000)\].

Thus, for creating a type-A descriptor, the database creator merely specifies an attribute (i.e., Salary) and a range of values ($2,000 - $10,000) for that attribute. We term the value to the left of the attribute the **lower limit** and the value to the right of the attribute the **upper limit**.

**Type-B:** The descriptor is an equality predicate. An example of a type-B descriptor is:

\[(\text{Position} = \text{PROFESSOR})\].

**Type-C:** The descriptor consists of only an attribute name, known as the **type-C attribute**. Let us assume that there are n different keywords $K_1, K_2, \ldots, K_n$, in the records of a database with a type-C attribute. Then, this type-C descriptor is really equivalent to n type-B descriptors $B_1, B_2, \ldots, B_n$, where $B_i$ is the equality predicate satisfied by $K_i$. In fact, this type-C descriptor will cause n different type-B descriptors to be formed. From now on, we shall refer to the type-B descriptors formed from a type-C descriptor as type-C
sub-descriptors. For instance, consider that Dept is specified as a type-C attribute for a file of employee records. Furthermore, let all employees in the file belong to either the TOY department or the SALES department. Then two type-B descriptors will be formed for this file. They are (Dept=TOY) and (Dept=SALES).

The database creator must observe certain rules in forming descriptors. These are specified below:

(1) Ranges specified in type-A descriptors for a given attribute must be mutually exclusive.

(2) For every type-B descriptor of the form (attribute-1 = value-1), no type-A descriptor can have the same attribute (i.e., attribute-1) and a range that contains its value (i.e., value-1).

(3) An attribute that appears in a type-C descriptor must not also appear in a type-A or a type-B descriptor defined previously.

(4) Type-A descriptors are specified first; Type-B descriptors next; Type-C descriptors last.

A keyword is said to be derived or derivable from a descriptor if one of the following holds:

(a) The attribute of the keyword is specified in a type-A
descriptor and the value is within the range of the descriptor.

(b) The attribute and value of the keyword match those specified in a type-B descriptor.

(c) The attribute of the keyword is specified in a type-C descriptor.

The use of these descriptors will be demonstrated in Chapter IV.

A query conjunction, or simply conjunction, is a conjunction of predicates. An example of a query conjunction is:

\[(\text{Salary} > 25000) \& (\text{Dept} = \text{TOY}) \& (\text{Name} = \text{JAI})\].

We say that a record satisfies a query conjunction if the record contains keywords that satisfy every predicate in the conjunction.

A query is any arbitrary Boolean expression of predicates. An example of a query is:

\[((\text{Dept} = \text{TOY}) \& (\text{Sal} < 100)) \lor ((\text{Dept} = \text{BOOK}) \& (\text{Sal} > 500))\].
3.2.2 The Data Manipulation Language (DML)

The data manipulation language for MDBS is a non-procedural language which supports four different types of requests - retrieve, insert, delete and update. The syntax of these various requests and examples of them are presented below. For a formal specification of DML, the reader may refer to Appendix A.

A. Retrieve

The syntax of a retrieve request is:

RETRIEVE Query Target-List [BY Attribute] [WITH Pointer]

That is, it consists of five parts. The first part is the name of the request. The second part is a query (as defined in Section 3.2) which identifies the portion of the database to be retrieved. The target-list is a list of elements. Each element is either an attribute, e.g., Salary, or an aggregate operator to be performed on an attribute, e.g., AVG(Salary). We will support five aggregate operators - AVG, SUM, COUNT, MAX, MIN - in MDBS. An example of a target-list of two elements is (Dept, AVG(Salary)). The values of an attribute in the target-list are retrieved from all records identified by the query. If no aggregate operator is specified on the attribute in the target-list, its values in all the records identified by the query are
returned directly to the user or user program. If an attribute is specified on the attribute in the target-list, some computation is to be performed on all the attribute values in the records identified by the query and a single aggregate value is returned to the user or user program. The fourth part of the request, referred to as the **BY-clause**, is optional as designated by the square brackets around it. The use of the By-clause is explained by means of an example. Assume that employee records are to be divided into groups on the basis of the departments for the purpose of calculating the average salary for all the employees in a department. This may be achieved by using a retrieve request with the specific target-list, (AVG(Salary)), and the specific BY-clause, **BY Department**. Finally, the fifth part of the request, which is also optional, is a WITH-clause which specifies whether pointers to the retrieved records must be returned to the user or user program for later use in an update request. Some examples of retrieve requests are presented below.

Example 1. Retrieve the names of all employees who work in the Toy Department.

```
RETRIEVE (File=EMP)&(Dept=TOY) (Name)
```
employees making more than $5000 per year.

RETRIEVE (File=EMP)&(Sal>5000) (Name,Sal)

Example 3. Find the average salary of an employee.

RETRIEVE (File=EMP) (AVG(Sal))

Example 4. List the average salary of all departments.

RETRIEVE (File=EMP) (AVG(Sal)) BY Dept

B. Insert

The syntax of an insert request is:

INSERT Record

where, Record is the record to be inserted into the database. An example of an insert command is:

INSERT (<Relation,EMPLOYEE>,<Salary,5000>,<Dept,TOY>)

C. Delete

The syntax of a delete request is:

DELETE Query

where Query is a query which specifies the particular records to be deleted from the database. An example of a DELETE request is:

DELETE (Name=HSIAO) V (Salary>50)
D. Update

In DML, the syntax of an update request is:

UPDATE Query Modifier

where Query specifies the particular records to be updated from the database and Modifier specifies the kinds of modification that need to be done on records that satisfy the query. An update in MDDBS which provides for the modification of only a single attribute value will have the attribute referred to as the attribute being modified. The modifier in an update request specifies the new value to be taken by the attribute being modified. The new value to be taken by the attribute being modified is specified as a function f of the old value of either that attribute (i.e., Type-I) or some other attribute (Types II, III and IV). The modifier may be one of the following five types:

TYPE 0: \(<\text{attribute}=\text{constant}>\>

TYPE-I: \(<\text{attribute}=f(\text{attribute})>\>

TYPE-II: \(<\text{attribute}=f(\text{attribute1})>\>

TYPE-III: \(<\text{attribute}=f(\text{attribute1})\text{ of Query}>\>

TYPE-IV : \(<\text{attribute}=f(\text{attribute1})\text{ of Pointer}>\>

Let a record being modified be referred to as the record being modified. Then, a Type-0 modifier sets the new value of the attribute being modified to a constant. A Type-I modifier sets the new value of the attribute being
modified to be some function of its old value in the record being modified. A Type-II modifier sets the new value of the attribute being modified to be some function of some other attribute value in the record being modified. A Type-III modifier sets the new value of the attribute being modified to be some function of some other attribute value in another record uniquely identified by the query in the modifier. Finally a Type-IV modifier sets the new value of the attribute being modified to be some function of some other attribute value in another record identified by the pointer in the modifier. An example of a Type-0 modifier is:

\[ <\text{Salary}=5000> \]

This sets the salary of all the records being updated to 5000.

An example of a Type-I modifier is:

\[ <\text{Salary}=1.1*\text{Salary}> \]

This raises the salary of all qualifying employees by 10%.

An example of a Type-II modifier is:

\[ <\text{Monthsal}=\text{Yearsal}/12> \]

This sets the monthly salary of all qualifying employees to be a twelfth of their own yearly salaries.
An example of a Type-III modifier is:

\(<\text{Salary}=\text{Salary of (Relation=WIFE)\&(Name=TARA)}\>\).

An example of a Type-IV modifier is:

\(<\text{Salary}=\text{Salary of 2000}\>\)

which modifies the salary of all qualifying employees to that of the record stored in location 2000. In order to use this type of modifier, the user must have previously issued a retrieve request with the WITH POINTER option.

An example of a complete update request would be:

\(\text{UPDATE (File=EMPLOYEE) } \langle\text{Salary=Salary+25}\rangle\)

which gives a $25 raise to all employees.

3.2.3 The Notion of a Transaction

In DML, we allow the flexibility for a user to specify a set of requests for repeated execution. Such a pre-specified set of requests shall be referred to as a transaction. As in other systems, a transaction must preserve consistency. A database-creator specifies a set of assertions on the database. These assertions are constraints which must be satisfied by data in the database. For instance, since employees cannot have negative salaries, an assertion on the database may require that all employees
have positive salaries. An assertion about a database is said to be true in the database if the data in the database satisfies the constraints in the assertion. A database is in a consistent state if all the assertions made on the database by the database-creator are true in the database. Finally, a transaction is said to preserve consistency, if the database is in a consistent state before it begins execution and the database is in a consistent state after it finishes execution.

Some examples of transactions in our environment are presented below. We begin each transaction with BOT, an acronym for beginning-of-transaction and we terminate each transaction with EOT for end-of-transaction.

Example 1. We assume the existence of two files in a database [Esua76], one called accounts and the other called assets as in Figure 12. The only assertion made by the database-creator on this database is that the total assets at a particular location be equal to the sum of the balances at that location. We assume that the record

\((<Loc, NAPA>, <Num, 5320>, <Bal, 287>)\)
<table>
<thead>
<tr>
<th>Location</th>
<th>Number</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAPA</td>
<td>32123</td>
<td>1050</td>
</tr>
<tr>
<td>ST HELENA</td>
<td>36592</td>
<td>506</td>
</tr>
<tr>
<td>NAPA</td>
<td>5320</td>
<td>287</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Location</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST HELENA</td>
<td>506</td>
</tr>
<tr>
<td>NAPA</td>
<td>1337</td>
</tr>
</tbody>
</table>

Figure 12. A Sample Database of Two Files
(Adopted from [Eswe76])
has to be inserted repeatedly into the database. Each time such an insertion takes place, the sum of the balances at location Napa is increased by $287. Hence, if the assertion that the sum of the balances at a location be equal to the total assets at that location is to be preserved, then the total for the location Napa in the assets file must also be increased by $287. A transaction that does this is:

BOT

INSERT (Loc:NAPA, Num:5320, Bal:287)

UPDATE (File=ASSETS) & (Loc=NAPA) <Tot=Tot+287>

EOT

Example 2. From time to time, employee Smith is given a $50 raise. An assertion on the database requires that employee Jones always make the same salary as employee Smith. Hence, the salary of Jones must also be raised by $50 whenever the salary of Smith is raised by $50. A transaction that does this is:

BOT

UPDATE (File=EMP) & (Name=SMITH) <Sal=Sal+50>

UPDATE (File=EMP) & (Name=JONES) <Sal=Sal+50>

EOT
3.2.4 In Meeting the Selection Criteria

In the beginning of this chapter, we stated three criteria which were to be satisfied by a data model and the data manipulation language of the data model. The three criteria were intended for the selection of an ideal data model and its data manipulation language forMDBS implementation. These criteria were referred to as the translation criterion, the partition criterion and the language criterion.

The attribute-based model satisfies the selection criteria as we amply argued in the preceding sections. In this section, we consider its data manipulation language, DML, in terms of the selection criteria.

Of the three criteria, the partition criterion is to be met by the model alone. Hence, we do not need to consider this criterion for DML. We shall consider the remaining criteria for DML in the following paragraphs.

In [Bane80], it was demonstrated that the data manipulation languages of the three prevailing data models—network, hierarchical and relational—could be translated into the data manipulation language of a database computer, known as DBC. The data manipulation language proposed in this chapter, i.e., DML, properly contains all the features
of the data manipulation language of DBC. Hence, if the data manipulation languages of the three prevailing data models can be translated into the data manipulation language of DBC, they can also be translated into DML. Thus, DML satisfies the translation criterion.

Finally, DML allows for the concurrent access of large volumes of data because it enables the user to specify queries in terms of boolean expressions of predicates. Consequently, DML satisfies the language criterion.

In conclusion, the proposed DML satisfies all the criteria indicated in the beginning of this chapter.
CHAPTER IV

THE PROCESS OF REQUEST EXECUTION

In Chapter III, we had described the data manipulation language, DML, used in MDBS. In this Chapter, we will describe the process of DML request execution from the time a DML request is first received by MDBS to the time the response set of the request is returned to the user or user program.

Central to the discussion on request execution is the notion of a cluster. This notion will be developed in Section 4.1. Here, we give a brief introduction to the process of request execution. We first utilize the descriptors defined in Chapter III as an equivalence relation to partition the database into equivalence classes.
which are termed clusters. The equivalence relation guarantees the following nice properties: Every record in the database belongs to one and only one cluster of the database. By proper use of the descriptors, the clusters may be formed in such a way that if a user needs to access a record belonging to a cluster, the user is most likely to have the need to access all the other records belonging to that cluster. Thus, clusters serve as the basic units of access in MDBS and every database is stored in MDBS as a collection of clusters.

The execution of a user request in MDBS proceeds typically in several distinct phases as follows. In the first phase, MDBS will determine the exact clusters of records which will satisfy the request. In the second phase, MDBS accesses security information about the user in order to select for the user the authorized clusters among the clusters which have been determined in the first phase. In the third phase, MDBS determines the secondary memory addresses of the authorized clusters selected in the second phase. In all these three phases, MDBS makes no access to the records of the clusters selected. Instead, MDBS utilizes auxiliary information about the clusters and security. Such utilization of auxiliary information constitutes the directory management function of MDBS. After the three phases of directory management, MDBS
retrieves the clusters of records which will satisfy the request.

The use of the data placement strategy in MDBS will ensure that each cluster (database) is stored in such a way that the records (clusters) of the cluster (database) are evenly distributed across the multiple back-ends of MDBS. Since all clusters are evenly distributed across the back-ends, the response set of the request will be retrieved in a parallel fashion from the back-ends.

This completes our brief description of the processing of a request in MDBS. The remaining sections of this chapter are organized as follows. In Section 4.1, we will develop the notion of a cluster. The data structures to be used for determining the set of clusters which will satisfy a request, and the algorithms necessary for such determination are also described in Sections 4.1. In keeping with our findings in Chapter II, each cluster is placed according to the track-splitting-with-random placement policy which was shown to be superior. In Section 4.2, we will describe two of the three phases of directory management. Phase 2 is dealing with security-related directory management. Because of its importance and our new contribution, it is discussed in Chapter V. Several strategies for performing directory management are proposed.
and evaluated on the basis of a criteria which strives for minimal processing time. These same strategies are also evaluated, in Section 4.2, in terms of storage requirements. At the end of Section 4.2, we will present our recommendations for a directory management strategy for MDBS. In Section 4.3, we will describe the execution sequences for the various DML requests that may be issued to MDBS.

4.1 The Notion of Record Clusters

Record clusters are formed for the purposes of narrowing the search space and minimizing the effort needed to retrieve records which may satisfy a given request. In other words, by organizing a database into clusters and by maintaining information about these clusters, MDBS may readily identify those clusters whose records will satisfy the given request, thereby achieving high throughput and good response time.

Although the notion of a record cluster for the above-mentioned purposes is well known the effectiveness of clusters for throughput gain and response time improvement lies in the effectiveness of the clustering algorithm for forming clusters and, more importantly, the placement
strategy for storing these clusters. In other words, it depends on how clusters are formed and placed. Interestingly enough, it does not depend on how clusters are used. In other words, the throughput and response time of MDBS are 'immune' from the way the clusters are utilized. This is because every request execution by MDBS will involve the search and retrieval of clusters. Such search and retrieval can always be shown to be maximal for throughput gain and response-time improvement. We will comment on this point briefly herein and elaborate on the point thoroughly in the later sections. Briefly, this is due to our use of the descriptors as a means to define and form clusters. As we recall from Chapter III, a descriptor is either a single predicate or a conjunction of predicates. We may also recall that a query in a user request is a boolean expression of predicates. Thus, a given user request will require the retrieval of data which satisfy the predicates of the expression. Since clusters are formed by the definition of descriptors and both descriptors and queries utilize the common notion of predicates, the data retrieved for the request are actually one or more clusters. Clusters therefore become the ideal formation (or unit) of data for storage and retrieval and for performance optimization.
Our work on clusters is unique also in a couple of respects. First, the clustering algorithms in our method will be executed in multiple back-ends rather than at a single back-end as in all other work. Second, we recognize the importance which must be given to the placement of these clusters across the multiple back-ends of MDBS.

In the following sections, we will describe how the clusters are formed in MDBS and how they are used. We will begin with some definitions.

4.1.1 Cluster Formation

For a database, the creator of the database specifies a number of descriptors called clustering descriptors, or simply, descriptors. An attribute that appears in a descriptor is called a directory attribute. We say that a directory attribute belongs to a descriptor if the attribute appears in that descriptor.

We recall that a record consists of attribute-value pairs or keywords. For purposes of clustering, only those keywords of the record which contain directory attributes are considered. Such keywords of the record are termed directory keywords. From the rules for forming descriptors specified in Chapter III, it is easy to see that a directory
keyword is derivable from at most one descriptor. For example, consider a database with Salary as the only directory attribute. Furthermore, let \((0 \leq \text{Salary} \leq 500)\) be the only descriptor \(D_1\) on Salary specified by the database creator. Now consider two records, one containing the directory keyword \(<\text{Salary}, 250>\) and the other containing the directory keyword \(<\text{Salary}, 750>\). Clearly, the former directory keyword is derivable from descriptor \(D_1\) and the latter directory keyword is not derivable from \(D_1\). Hence, the latter keyword is not derivable from any descriptor in the database and we say that the directory keyword is derivable from no descriptor. Since a record may have many directory keywords each of which will be derivable from at most one descriptor, we say that the record is derived from a set of descriptors. It is possible for a record to be derived from the empty set of descriptors. There are two such cases. In the first case, it may happen that a record does not contain any directory keyword. In this case, it is said that the record is derived from the empty set of descriptors. Thus, going back to the previous example with the single directory attribute, Salary, and the single descriptor, \((0 \leq \text{Salary} \leq 500)\), a record which does not contain any salary information (i.e., no keyword with the attribute Salary) is said to be derived from the empty set of descriptors. The second case in which a record is derived
from the empty set of descriptors is when the record does
indeed contain directory keywords, but these keywords are
not derivable from any descriptors. In the previous
example, a record with the directory keyword <Salary,750>
which is not derivable from the descriptor is therefore
derived from the empty set of descriptors also.

If two records are derived from the same set of
descriptors, they are likely to be retrieved together in
response to a user request, since these two records have
keywords which are derived from the same set of descriptors.
Thus, these two records should be stored together in the
same cluster. A cluster is, therefore, a group of records
such that every record in the cluster is derived from the
same set of descriptors. We say that a record cluster is
defined by the set of descriptors from which all records in
the cluster are derived.

It is easy to see that a record belongs to one and only
one cluster. The reasoning is as follows. A record
consists of zero or more directory keywords. If it consists
of zero directory keywords, it belongs to the cluster
declared by the empty set of descriptors. If the record
consists of one or more directory keywords, then, the record
must be derived from one and only one set of descriptors,
since each directory keyword is derived from at most one
descriptor. This unique set of descriptors defines the unique cluster to which the record belongs. Thus, we have used the concept of descriptor sets to partition the database into equivalence classes, namely clusters. A formal proof of the above observations is included in Appendix B.

In order to form clusters for the records in a database, an algorithm is provided herein which will take a record and determine its cluster. We will describe this algorithm informally below. The detailed algorithm to be implemented in MDBS for cluster formation will be presented in Appendix C.

The algorithm for determining the cluster to which a record belongs is as follows. For each attribute-value pair in the record, determine if the attribute is a directory attribute. If it is not, then that attribute-value pair is not used for cluster determination. If the attribute is a directory attribute, determine the descriptor, if any, from which it is derived. We refer to this descriptor, if any, as the corresponding descriptor for the given attribute-value pair. The set of corresponding descriptors for all the attribute-value pairs in a record defines the cluster to which the record belongs.
By using the algorithm on every record of a database at database-creation time, we may form the record clusters of the database.

4.1.2 An Example of Cluster Formation

Consider the database and its descriptors shown in Figure 13. The figure shows two files, accounts and assets. The accounts file has three records and the assets file has two records. Also, the figure shows the five descriptors specified on this database by the database-creator. The attributes Number and Balance, are type-A descriptors and the attribute, Location, is a type-C descriptor. This type-C descriptor will be converted into two type-B descriptors as shown in Figure 14. There is no descriptor for the attribute, Total, because no request with the attribute, Total, as part of a query is expected. The clusters formed for this database and the records in each cluster are depicted in Figure 15.

The first row in Figure 15 is for cluster 1. The set of descriptors defining this cluster is (D2, D3, D5). Consider record R1 which belongs to this cluster. It contains the keyword <Location, NAPA> which is derived from the descriptor D5 (Location=NAPA). Similarly, the keyword
Accounts file:

- (<Location, NAPA>, <Number, 32123>, <Balance, 50>)
- (<Location, St. Helena>, <Number, 5320>, <Balance, 506>)
- (<Location, NAPA>, <Number, 36592>, <Balance, 287>)

Assets file:

- (<Location, St. Helena>, <Total, 506>)
- (<Location, NAPA>, <Total, 337>)

The database creator specifies the following descriptors:

Type-A Descriptors:

1. \(0 \leq \text{Number} < 15000\)
2. \(15,000 \leq \text{Number} < \infty\)
3. \(0 \leq \text{Balance} < 500\)
4. \(500 \leq \text{Balance} < 1000\)

Type-C Descriptor: Location

* For simplicity, we refer to the three records of the accounts file as R1, R2 and R3, respectively; and to the two records of the assets file as R4 and R5, respectively.

Figure 13. A Database of Two Files and its Clustering Descriptors
Descriptors Specified by the Database-Creator as seen by MDBS

<table>
<thead>
<tr>
<th>Descriptor</th>
<th>Descriptor id</th>
</tr>
</thead>
<tbody>
<tr>
<td>((0 \leq \text{Number} &lt; 15,000))</td>
<td>D1</td>
</tr>
<tr>
<td>((15,000 \leq \text{Number} &lt; \infty))</td>
<td>D2</td>
</tr>
<tr>
<td>((0 \leq \text{Balance} &lt; 500))</td>
<td>D3</td>
</tr>
<tr>
<td>((500 \leq \text{Balance} &lt; 1000))</td>
<td>D4</td>
</tr>
<tr>
<td>((\text{Location} = \text{Napa}))</td>
<td>D5</td>
</tr>
<tr>
<td>((\text{Location} = \text{St. Helena}))</td>
<td>D6</td>
</tr>
</tbody>
</table>

The Descriptors Formed for the Database of Figure 13

Figure 14. The Descriptor-to-Descriptor-Id Table (DDIT)
The Clusters Formed for the Database of Figure 13

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Set of Descriptors*</th>
<th>Records in the Cluster Defined by the Descriptor Set**</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{D2, D3, D5}</td>
<td>R1, R3</td>
</tr>
<tr>
<td>2</td>
<td>{D1, D4, D6}</td>
<td>R2</td>
</tr>
<tr>
<td>3</td>
<td>{D6}</td>
<td>R4</td>
</tr>
<tr>
<td>4</td>
<td>{D5}</td>
<td>R5</td>
</tr>
</tbody>
</table>

* Actually, only descriptor ids are shown here. Together with the Descriptor-To-Descriptor-Id Table shown in Figure 14, MDBS maintains the descriptor sets.

** In implementation, this column contains the secondary memory addresses of R1, R2, R3, R4 and R5 with two addresses in the first row.

Figure 15. The Cluster-Definition Table (CDT)
<Number, 32,23> is derived from D2 which is \((15000 \leq \text{Number} < \infty)\). Finally, the keyword <Balance,50> is derived from D3 which is \((0 \leq \text{Balance} < 500)\). Hence, R1 belongs to the cluster defined by D2, D3 and D5. Similarly, it may be shown that R3 belongs to the same cluster. This explains the first row in Figure 15. With similar exercises, we may verify the remaining three rows in Figure 15.

4.1.3 Clusters Determination During Request Execution

Up to this point, we have been describing the process of cluster formation. We will now explain how clusters are used during request execution. More specifically, we will explain how to determine the cluster to which a new record belongs and how to determine the set of clusters which must be retrieved in order to satisfy a query for retrieval, deletion or update.

During the process of cluster formation described in the previous section, MDBS uses an algorithm repeatedly for determining the cluster of a record in the database. This same algorithm may now be used by MDBS to determine the cluster of a record for the record insertion. In insertion, the cluster definition table (CDT) is used in order to
determine the secondary memory address (addresses) of this cluster.

Next, let us describe how MDBS determines the set of clusters which satisfy the query in a retrieval, deletion or update request. Before we may do this, we must introduce some concepts and terminology.

Descriptor \( w \) is defined to be less than descriptor \( Y \), if the attributes in both descriptors are the same and one of the following holds.

1. Both descriptors are of type-A and the upper limit of descriptor \( w \) is lower than the lower limit of descriptor \( Y \).
2. Both descriptors are of type-B and the value in descriptor \( w \) is smaller than the value in descriptor \( Y \).
3. Descriptor \( w \) is of type-A and descriptor \( Y \) is of type-B and the upper limit of descriptor \( w \) is lower than the value in descriptor \( Y \).
4. Descriptor \( w \) is of type-B and descriptor \( Y \) is of type-A and the value in descriptor \( w \) is smaller than the lower limit of descriptor \( Y \).

An exactly parallel description for the greater-than relation among descriptors may also be given. The above definition covers the case where either \( w \) or \( Y \) is a type-C descriptor, since type-C descriptors are stored as type-B
descriptors in MDBS

To illustrate the definition of less-than among descriptors, let us assume that we are given the descriptors D1 (100 ≤ Salary < 200), D2 (0 ≤ Salary < 99), D3 (Salary = 99) and D4 (Salary = 200). Thus, D3 is less than D1, D2 is less than D3, and D1 is less than D4. The relation, less-than, is transitive. Hence, we can define a partial ordering among descriptors. For example, the ordering among these four descriptors is D2, D3, D1 and D4.

Using the above definition of less-than and greater-than for the descriptors, we are ready to describe the algorithm for determining the corresponding set of clusters for a query in a user request. The query is assumed to be in disjunctive normal form, i.e., disjunction of conjunctions. The algorithm will proceed in three steps.

Since a query conjunction consists of predicates, we will determine, in the first step, a corresponding descriptor or a corresponding set of descriptors for each predicate. This is done as follows. If the predicate in a query conjunction is an equality predicate, then the corresponding descriptor is the one from which the keyword satisfying the predicate is derived. For example, if the predicate is Location=NAPA, then the keyword satisfying the predicate is <Location, NAPA> and the corresponding
descriptor is (Location=NAPA). If the predicate is either a less-than or less-than-or-equal-to predicate, it is first treated as an equality predicate and the corresponding descriptor D for that equality predicate is first determined. Then, all the descriptors less than D, along with D, form the corresponding set of descriptors for the less-than or less-than-or-equal-to predicate. If the predicate is a greater-than or greater-than-or-equal-to predicate, then it is first treated as an equality predicate and the corresponding descriptor D for that equality predicate is first determined. Then, all the descriptors greater than D, along with D, form the corresponding set of descriptors for the greater-than or greater-than-or-equal-to predicate. Thus, we have determined a corresponding set of descriptors for a predicate.

The above procedure is repeated for every predicate in the query conjunction. Thus, we will have determined a corresponding set of descriptors for every predicate in a query conjunction.

Our next step is to determine the corresponding set of clusters for a query conjunction, since a query consists of one or more query conjunctions. Let the query conjunction have p predicates. Let the set of descriptors corresponding to the i-th predicate be Si. Now, form all possible groups,
where each group consists of one descriptor from $S_i$ for $i$ ranging from 1 to $p$. In other words, we are forming the cross-product of $S_i$. The reason for forming this cross-product of $p$ sets is because a query conjunction consists of a conjunction of $p$ predicates, each of which has a corresponding set $S_i$ of descriptors. Each element in this cross-product is termed a descriptor group which is of course a set of descriptors. Intuitively, a group defines a set of clusters whose records satisfy the query conjunction.

We recall that MDBS maintains a table, known as the cluster definition table, which is created at the database creation time. (See Figure 15 again for an example). However, the definitions kept in the table may not be identical to the definitions of the groups. Without relating the descriptor groups with the descriptor sets kept in the table, we may not be able to determine the clusters involved. Thus, this second step includes the determination of whether there are descriptor sets in the table which contain a descriptor group. If there are such sets, then the clusters defined by the descriptor sets are indeed the clusters referred to by the descriptor group.

By repeating this procedure for every descriptor group in the cross-product, we are able to determine the corresponding set of clusters for a query conjunction. The
entire second step which is used to determine the corresponding set of clusters for a query conjunction is then repeated for every query conjunction in the query. Thus, we have determined a corresponding set of clusters for every query conjunction in the query.

The final step of the algorithm determines the corresponding set of clusters for the query from the corresponding set of clusters for each query conjunction in the query. Since the query is a disjunction of conjunctions, the corresponding set can be simply obtained as the union of the sets of clusters for each query conjunction in the query.

The three steps involved in this algorithm are illustrated with an example in the following section and formally specified in Appendix C.

4.1.4 An Example of Clusters Determination During Request Execution

Our example is developed around the database and descriptors of Figures 13 and 14. Consider that the following retrieve request is received by MDBS.

RETRIEVE (Loc=St.Helena)&(Bal=506) V (Num<5500)(Bal)
Clearly, the corresponding descriptor for the predicate (Location=St. Helena) is D6 and that for the predicate (Balance=506) is D4. Similarly, the corresponding descriptor for the predicate (Number<5500) is D1. Thus, we complete the first step of the algorithm for determining the corresponding set of descriptors for each predicate.

In the next step, we need to determine the corresponding set of clusters for each query conjunction. Consider the first query conjunction, (Location=St. Helena) & (Balance=506). The only descriptor group that can be formed for this query conjunction is (D6, D4). In searching the entries of the descriptor definition table depicted in Figure 15, we discover that there is only one descriptor set which contains the descriptor group. It is the set (D1, D4, D6). The cluster defined by (D1, D4, D6), i.e., cluster 2, is the only member of the corresponding set of clusters for the first query conjunction. Similarly, we determine the corresponding set of clusters for the second query conjunction, (Number<5500). It so happens that cluster 2 is also the only member of the corresponding set of clusters for the conjunction. Thus, we complete the second step of the algorithm.
In the final step of the algorithm, the union of all members of the corresponding sets of clusters of the query conjunctions is still cluster 2. Thus, cluster 2 constitutes the only member of the corresponding set of clusters for the given query. Once the corresponding set of clusters is determined, the addresses of the records in the clusters may be used for access to the secondary memory.

4.2 Directory Management

The entire sequence of actions taken by MDBS from the time a record-insertion request is received to the time that the cluster to which the record is to be inserted is determined (i.e., the secondary memory address or addresses are generated) is referred to as directory management for an insert request. Similarly, the entire sequence of actions taken by MDBS from the time a retrieve, delete or update request is received to the time the corresponding set of clusters for the query in the request and their addresses in the secondary memory are generated is referred to as directory management for a non-insert request. Together, they constitute the directory management of MDBS.
We repeat that the directory management in MDBS consists of three major phases. In the first phase, MDBS determines the exact clusters of records which will satisfy the user request. This phase was described in the previous section. The algorithm used in this phase for determining the clusters was briefly described in the previous section and in detail in an appendix. In the second phase, MDBS accesses security information about the user in order to select for the user the authorized clusters among the clusters which have been determined in the first phase. This discussion is relegated to Chapter V. In the third phase, MDBS determines the secondary memory addresses of the authorized clusters selected in the second phase.

4.2.1 Two phases of Processing - Descriptor Processing and Address Generation

For implementation of the directory management function, we recall that the cluster definition table CDT is used to determine the corresponding set of clusters for a query. At the same time, the secondary memory addresses of the corresponding set of clusters may also be found, since these addresses are present in the third column of CDT (see Figure 15 again). However, we want to ensure that only the secondary memory addresses of the authorized clusters for a
user are utilized. This is achieved by augmenting the CDT of Figure 15 with several more columns of security-related information, one column for each user of the database. The details of the kinds of security-related information maintained for each user are given in Chapter V. Thus, in implementation of the directory management function, the three phases described above may be elaborated as follows.

In the first phase, the descriptor-to-descriptor-id table DDIT is searched to determine the corresponding descriptor or descriptors for each predicate of a query in the case of a non-insert request and for each keyword of a record in the case of an insert request. In the sequel, we shall refer to this phase as the descriptor search phase and we shall refer to the processing performed therein as descriptor processing. In the next step, the augmented CDT is searched and the corresponding single cluster in the case of an insert request or the corresponding set of clusters in the case of a non-insert request is determined. Once the cluster or cluster set is determined, the authorized cluster or clusters may be selected on the basis of security-related information in the augmented CDT. By searching the same augmented CDT the addresses of authorized cluster(s) can be found. We shall refer to this step, in the sequel, as the address generation phase and we shall refer to the processing performed therein as address generation. Thus,
the three phases of directory management are now consolidated in two.

Since the descriptor search phase and the address generation phase are similar for both insert and non-insert requests, in the sequel we shall only consider these two phases for non-insert requests. The discussion extends in a straightforward manner to insert requests.

4.2.2 Processing Strategies for Multiple Back-ends

In previous discussions, we make no distinction whether the two phases were to be carried out in a single computer or in multiple computers (a controller and several back-ends). It is now necessary to discuss how these two phases will be executed in the controller and multiple back-ends of MDBS. We have identified for MDBS six different strategies for carrying out the descriptor search phase in the multiple back-ends and one strategy for carrying out the descriptor search phase in the controller. Thus, a total of seven strategies are identified for various ways of carrying out the descriptor search phase. We have also identified two strategies, one for carrying out the address generation phase in the controller, the other in the back-ends. For completeness, we also consider an eighth
strategy for directory management in which both phases are carried out in the controller. These eight strategies for directory management are proposed and evaluated in the next sections.

Let us first, however, indicate our preference for a strategy in which the address generation phase is carried out in multiple back-ends rather than in the controller. By carrying out this phase in the back-ends, MDBS will be alleviated from the controller limitation problem. Since this phase deals with the generation of secondary memory addresses, each back-end would need to generate only those secondary memory addresses associated with that back-end. On the other hand, if the addresses were to be generated by the controller, the controller would need to generate all the relevant secondary memory addresses associated with the entire back-ends. Thus, it is easy to see that the work of address generation is evenly divided up among the back-ends in the former case. This is essential if we are to achieve an ideal system in which the response time is inversely proportional to the number of back-ends. This concludes our discussion of our preference for a strategy in which the address generation phase is carried out in the multiple back-ends.
We note that the address generation phase actually includes all security related processing also. The time for address generation, which is essentially the time for searching the augmented CDT, will depend on the size of each entry in the augmented CDT. Hence, it will depend on whether or not the table is augmented with security information. In the sequel, our analysis will assume that no security information is contained in the CDT. There are two reasons for this assumption.

First, we wish to analyze the performance of an MDBS in which security is not enforced. This is because many implementations of MDBS may wish to provide only the basic database management functions and may not wish to provide security enforcement.

Second, the enforcement of security will not affect our comparative study of the various strategies, since all the back-ends are expected to spend the same amount of work in security-related processing.

We now proceed to discuss the eight different strategies for directory management in the following sections. We propose various strategies for directory management in terms of descriptor searching and address generation. These alternatives will be evaluated from the viewpoint of performance and storage requirements.
A. The Centralized Strategy

In this strategy, all the directory management is done at the controller. The controller maintains the descriptors for all the directory attributes. In other words, both the DDIT and the augmented CDT are stored with the controller. Given a query, the controller first performs the descriptor processing and then address generation by utilizing the aforementioned tables. Eventually, a set of secondary memory addresses of relevant records is generated at the controller. We note that a secondary memory address consists of not just the track and cylinder information about the records but also the information about the back-end in which the track and cylinder are located. None of the remaining seven strategies will need the back-end information in a secondary memory address. This is because all the remaining strategies will do the address generation at the respective back-end rather than at the controller.

B. The Partially Centralized Strategy

The descriptor processing is done at the controller in this strategy as in the previous strategy. The corresponding descriptors are then broadcasted to all the back-ends. Each back-end will now carry out the address
generation phase in an independent fashion. The reason why we expect this strategy to be superior to the previous strategy is two-fold. First, the work of address generation that could be done at the controller is now distributed to the back-ends. This should alleviate the controller limitation problem. Second, the work needed for address generation is divided in such a way that a back-end needs only to generate the addresses of the secondary memory of that back-end.

In this strategy, the descriptor search phase of the directory management is still performed at the controller. The six remaining strategies we will consider are those which try to diminish the effect of having this descriptor search phase performed solely at the controller. In other words, the following strategies will alleviate the controller limitation problem further.

C. The Rotating Strategy

In the rotating strategy, the descriptor processing during the descriptor search phase is done in a round-robin fashion among the controller and the back-ends. More specifically, the first query is processed at the controller, the second query is processed at the first
back-end, the third query is processed at the second back-end, and so on. As a result, it is hoped that some alleviation of the controller limitation problem will take place. As in the partially centralized strategy the address generation is done individually at each back-end. When the arrival rate of requests to MDBS is low, this strategy will not perform any better than the partially centralized strategy. On the other hand, when the arrival rate of request is high, this strategy may lead to an improvement in performance over the partially centralized strategy. This is because the descriptor processing on a number of queries by multiple back-ends may be overlapped while the descriptor processing on individual queries by the controller must be done sequentially in the partially centralized strategy.

D. The Rotating Without Controller Strategy

This strategy is very similar to the previous one. The only difference is that no descriptor processing is done at the controller. Thus, the descriptor processing for the first query is done at the first back-end, the descriptor processing for the second query is done at the second back-end, and so on. After descriptor processing is completed at a back-end, the corresponding descriptors must
be broadcast to all back-ends so that they may proceed with the address generation phase. The only reason for introducing this strategy into consideration is that it would appear to alleviate the controller limitation problem completely from directory management.

We note that both the rotating with controller and the rotating without controller strategies require the duplication of the necessary tables (i.e., DDIT) at all back-ends. This is tolerated for the following reasons. It will be shown, later, that the tables needed for address generation are very much larger than the tables needed for descriptor processing. As a result, duplicating the tables needed for descriptor processing for multiple back-ends is tolerable. Furthermore, we are willing to sacrifice storage, if it means an improvement in performance.

Up to this point, two of the strategies allow queries to be processed parallelly in the descriptor search phase. There is no parallel descriptor processing of predicates for a given query. In the following three strategies, we explore the possibility of parallel descriptor processing of predicates for individual queries during the descriptor search phase.
E. The Fully Duplicated Strategy

In the fully duplicated strategy, the descriptor processing is done across the back-ends. More specifically, if there are \( n \) back-ends in MDBS and a query contains \( x \) predicates, each back-end will process \( \frac{x}{n} \) predicates and generate \( \frac{x}{n} \) corresponding descriptors which will, in turn, be communicated to each other. Each back-end may then proceed to the address generation phase. Such a strategy also requires the necessary tables to be duplicated at each back-end.

F. The Descriptors Dividing by Attribute Strategy

In this strategy, we explore the possibility of achieving parallel descriptor processing without the need for any duplication of the necessary tables. If there are \( i \) directory attributes and \( n \) back-ends in MDBS, each back-end will maintain the descriptors corresponding to \( \frac{i}{n} \) attributes. Each back-end will process those predicates in a query where the descriptors corresponding to the attribute in the predicate is maintained at that back-end. It is expected that each back-end will do an equal amount of descriptor processing, although there may be cases where one back-end does more descriptor processing than the others.
This happens if all the predicates in a query are such that the descriptors that need to be searched are all stored at the same back-end.

G. The Descriptors Division Within Attribute Strategy

Like the previous strategy, this strategy also attempts parallel descriptor processing without any duplication of the necessary tables. If there are 1 descriptors on each directory attribute, each back-end will maintain for each attribute 1/n descriptors. Thus, descriptor processing is spread over the back-ends. All back-ends will participate in the descriptor processing of a query. After each back-end obtains some results, they exchange their results. Then, each back-end proceeds with its own address generation phase.

H. The Fully Replicated Strategy

This is the final strategy we consider for doing directory management. In this strategy, each back-end will work on the entire query during the descriptor search phase. The advantage of letting each back-end do the descriptor processing on all queries is that, unlike the previous three
strategies, exchanges among back-ends are unnecessary in this strategy because all back-ends have all the needed results. After completing the descriptor processing, each back-end does its address generation.

4.2.3 Performance Evaluation of the Directory Management Strategies

In this section, we compare the eight strategies for directory management on the basis of performance. For our convenience, we name these strategies A through H, respectively.

Two different approaches are used for the performance analysis. The first approach considers the directory management process alone. It does not consider the fact that there may be queues at the various back-ends and that the controller may allow overlapping of query handling for directory management. Specifically, the different strategies are compared in terms of the time duration from the receipt of a request to the point where all the necessary secondary memory addresses are generated, including the time taken for messages exchanges.
The second approach studies the relative response time for a typical request when each of these eight strategies is employed for directory management. In other words, the directory management is considered in terms of its effect on the overall response time of a request. This study employs a closed queueing network model of MDBS.

One issue that is important to both approaches is whether the necessary tables for descriptor processing and address generation can be stored in the main memory. In our analysis, we assume that a page of the descriptor-to-descriptor-id table DDIT is in the main memory with probability \( p \), where \( p \) is high. This is because DDIT is small. For the augmented cluster definition table, i.e., the augmented CDT, on the other hand, we assume that only a certain amount of the main memory, \( m \), is reserved for the table. Now, if the augmented CDT requires greater amount of memory, \( g \), then pages of the augmented CDT are assumed to be in the main memory with probability \( m/g \). In general, \( m/g < p \).

Another issue to be resolved before we begin our analyses concerns the searching of the descriptors. It is clearly possible to store the descriptors in sorted order and search for the right descriptor using a binary search. Another technique would be to store the descriptors as a B-tree, whose leaf nodes are the descriptors themselves.
However, since the number of descriptors is not expected to be very large, we shall assume that a binary search is used.

A. Time Analyses and Performance Equations

In this section, we present the results for the execution time of the directory management algorithms from the point that a query is received to the point where the addresses of the records are generated and made available at the back-ends. This time therefore includes the time for descriptor processing and address generation.

The following observation may be used to simplify our calculations. First, the centralized strategy and the partially centralized strategy differ only in the address generation phases. Clearly, the partially centralized strategy takes less time for address generation and, hence, is superior to the centralized strategy. We only need to compare the remaining strategies to determine the best one. All the remaining strategies use exactly the same algorithm for address generation. Hence, the time for address generation need not be included in our comparison study. With these observations, let us proceed with our comparison study. Let,

\[ t_m : \text{time to send a message from (to) the controller} \]
D : total number of descriptors
i : total number of directory attributes
td : time to read a descriptor from the main memory
tp : time to read a entry from the PT (see Appendix C) in the main memory.
ta : time to read a predicate
x : number of predicates in a query conjunction
tpr : time for an arithmetic operation
k : number of descriptors per secondary memory page
tpg : time to access a secondary memory page
lg : logarithm of the base 2
u(z): the nearest integer greater than or equal to z

In the ensuing discussions let us calculate the total time taken in order to complete the descriptor search phase and have the corresponding set of descriptors available at all back-ends. That is, let us calculate the time taken for directory processing and the time for exchanging any messages that may be needed. For simplicity, we assume that users only employ single conjunctions in their requests. If a disjunction of p conjunctions is used in a request, it may be easily treated as p separate requests.
In Strategies A, B, C, D and H, it is easy to see that
the descriptor processing on the entire query conjunction is
done at a single computer. Thus, the average-case time is
expressed as follows

\[ x(t_a + (1/2)(t_p + t_{pr}) + \log((t_d + 4t_{pr})D/i) + 2t_{pr}) - - (1) \]

Here is the explanation. For each of the \( x \) predicates in
the conjunction, the following must be done. First, the
predicate must be read and this takes \( t_a \) time units. Next,
the \( i \) entries in \( PT \) must be searched. On the average, \( 1/2 \)
entries will need to be searched. Each of the \( 1/2 \) entries
must be read (taking \( t_p \) time units each time) and compared
with the attribute in the predicate (taking \( t_{pr} \) time units).
Next, the set of \( D/i \) descriptors must be searched. Our
algorithm for a binary search of \( N \) descriptors is included
in Appendix D. From the algorithm, the total time to do
binary search on \( N \) descriptors is

\[ 2t_{pr} + \log(N) (\text{time to read a descriptor} + 4t_{pr}) \]

Thus, the time taken for doing a binary search on \( D/i \)
descriptors will take

\[ 2t_{pr} + \log((t_d + 4t_{pr})D/i) \]
Together, we obtain equation (1).

In arriving at equation (1), we assumed that all the descriptors are in the main memory. Let us now improve equation (1) by assuming that only a fraction $p$ of the descriptor pages are in the main memory. We assume that all the $D$ descriptors are stored in secondary-memory pages each of which contains up to $k$ descriptors. Also, descriptors for different attributes are stored in different pages. In searching descriptors organized as pages, we assume the following algorithm is used. Pages are retrieved sequentially. For each page retrieved, the ranges of the first and last descriptors in the page are compared to the value in the predicate. This will tell us if the page contains the descriptor we are looking for. If so, a binary search of that page is performed. If not, the next page is retrieved, and so on. Since the first page must be processed before the second one is brought in and because the pages needed may not be adjacent in the secondary memory, we assume no I/O overlap with CPU processing. Then, the time for descriptor processing is:

$$x(t_a + (1/2)(t_p + t_{pr}) + (u(D/(2ik))) - 1)$$

$$((1 - p)t_{pg} + 2(t_d + t_{pr})) +$$

$$((1 - p)t_{pg} + 2(t_d + t_{pr}) + t_b)) \quad \text{(2)}$$
in the average case. The worst-case time may be obtained from equation (2) by replacing \( u(D/(2ik)) \) with \( u(D/(ik)) \) and \( (i/2)(tp + tpr) \) with \( l(tp + tpr) \). Here, the time taken for doing a binary search on \( k \) descriptors, \( tb \), is \( (2tpr + \lg k (td + 4tpr)) \).

Let \( tdr \) be the time taken for descriptor processing. Then \( tdr \) is equal to equation (2) above. Finally, the time taken for descriptor processing and message exchanging in strategies A, B and H is

\[
(tm + tdr)
\]

where \( tm \) is the time taken to broadcast the corresponding descriptors from the controller to the back-ends in strategies A and B and it is the time to broadcast the original query conjunction to all the back-ends each of which individually spends \( tdr \) time units to calculate the corresponding descriptors in Strategy H.

In Strategy C, the time for descriptor processing and message exchanging is

\[
(n/(n + 1)) \ast (tdr + 2tm) + (l/(n + 1)) \ast (tdr + tm)
\]

This is explained as follows for a system with \( n \) back-ends.
If the processing is done in the controller and happens once every \((n+1)\) times, the processing must include the time for broadcasting the corresponding descriptors from the controller to the back-ends. On the other hand, if the processing is done at one of the back-ends as happens \(n\) out of \((n+1)\) times, then the processing time must include the time for sending the original query conjunction to a back-end and the time for broadcasting the corresponding descriptors to all back-ends from that back-end.

Finally, in Strategy D, the time for descriptor processing and message exchanging is

\[(tdr + 2tm)\]

We now need to calculate the descriptor processing time for strategies E, F and G. For Strategy E, the time for descriptor processing in the average case is:

\[u(x/n)(ca + (1/2)(tp + tpr)
+ ((1 - p)tpg + 2(td + tpr) + tb)
+ (u(D/2ik)) - 1)((1 - p)tpg
+ 2(td + tpr))) + 2tm - \text{--------(3)}\]

The time for descriptor processing in the worst case may be
obtained from equation (3) by replacing \((i/2)(tp + tpr)\) with \(i(tp + tpr)\) and \(u(D/(2ik))\) with \(u(D/(ik))\). Time for two message exchanges is also included in the above equation. One of the messages is for broadcasting the original query conjunction from the controller to all the back-ends. The second one is for exchanging partial results among the back-ends. We note that this method requires the presence of all the descriptors at each back-end. Furthermore, no more than \(x\) back-ends can be executing a query conjunction simultaneously, since there are \(x\) predicates in the query conjunction. This is the maximum degree of parallelism that can be brought to bear on the processing of descriptors.

Let us now present the descriptor processing and message exchange time for Strategy F as follows.

\[
u(x/n)(ta + u(i/(2n))(tp + tpr) \\
+ ((1 - p)tpg + 2(td + tpr) + tb) \\
+ (u(D/(2ik)) - 1)((1 - p)tpg \\
+ 2(td + tpr))) + 2tm - ----- ---(4)
\]

The worst case time is obtained from equation (4) by replacing \(u(x/n)\) with \(x\), \(u(i/(2n))\) with \(u(i/n)\), and \(D/(2ik)\) with \(D/(ik)\). In general, however, we expect this strategy to perform at the time closer to the average-case time than the worst-case time. This is because the database
administrator may use his knowledge of the typical query conjunctions to assign descriptors to back-ends in an appropriate fashion.

Finally, the average-case time for descriptor processing and message exchange by Strategy G is as follows.

\[ x(t_a + u(i/2)(t_p + t_{pr}) \\
+ ((1 - p)t_{pg} + 2(t_d + t_{pr}) + t_b) \\
+(u(D/(2nk))) - 1)((1 - p)t_{pg} \\
+ 2(t_d + t_{pr}))) + 2tm \]  

The worst-case time is obtained from equation (5) by replacing \( u(i/2) \) with 1 and \( D/(2nk) \) with \( D/(nk) \). It is clear that for small values of \( D \) and large values of \( k \), increasing \( n \) is going to have little effect in reducing the time. For instance, consider that the number of descriptors, \( D \), is 100 and that the number of descriptors stored in a page, \( k \), is 100. Then \( u(D/(nk)) = 1 \), irrespective of the number of back-ends, \( n \). Since \( u(D/(nk)) \) is the only expression in equation (5) which contains \( n \), it is clear that increasing \( n \) is not going to reduce the time for descriptor processing and message exchange. However, this strategy should prove advantageous for large values of \( D \).
B. Computations and Their Interpretations Resulted from the Performance Equations

In order to compare the descriptor processing and message exchange times of MDBS under various strategies we use the following values for the parameters in the equations (1) through (5) to calculate the times.

\[ t_m = 8 \text{msecs} \] (time to send a message from (to) the controller)
\[ t_{pr} = 5 \text{usecs} \] (time for an arithmetic operation)
\[ t_d = 5 \text{usecs} \] (time to read a descriptor from the main memory)
\[ t_{a} = 3 \text{usecs} \] (time to read a predicate)
\[ t_{p} = 3 \text{usecs} \] (time to read an entry from the DDIT in the main memory)
\[ t_{pg} = 47.3 \text{msecs} \] (time to access a secondary memory page)
\[ k = 85 \] (no. of descriptors per secondary memory page)
\[ p = 0.9 \] (probability of a page of DDIT entries being in the main memory)
\[ x = 5 \] (no. of predicates in a query conjunction)

The value of \( n \), the number of back-ends, is chosen from the set \((2, 5, 8)\). The ratio \( D/i \), the number of descriptors per directory attribute, is chosen from the set \((10, 20, 30, 40)\). The number of directory attributes is chosen from the
set (5, 10, 15).

The results of our calculations are shown in Figure 16. In this figure, we present two rows of results for each number of back-ends. The first row gives the average-case times and the second row gives the worst-case times for descriptor processing and message exchanging. It is seen that two of the three strategies that utilize parallel processing of predicates of a given query conjunction during the descriptors search phase, namely strategies E and F, give the best average-case results. They consistently outperform all the other strategies over the entire range of variation which we tried for the number of back-ends, the number of directory attributes and the number of descriptors per attribute. The time under strategies E and F may be as much as 12 msecs less than the time under the next best strategy. This occurs when the number of back-ends employed is eight.

Looking at the worst-case results, we see that strategy E is, once again, the best strategy. The worst-case performance of Strategy F, however, is far worse than that of Strategy E. This is because the worst case for Strategy F occurs when all the predicates in a query conjunction are such that the descriptors that need to be searched for descriptor processing are all stored at a single back-end.
<table>
<thead>
<tr>
<th>Strategy</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Back-ends</td>
<td>2</td>
<td>32.74</td>
<td>32.74</td>
<td>38.07</td>
<td>40.74</td>
<td>30.84</td>
<td>30.82</td>
<td>40.74</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>32.74</td>
<td>32.74</td>
<td>39.40</td>
<td>40.74</td>
<td>20.95</td>
<td>20.93</td>
<td>40.74</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>32.74</td>
<td>32.74</td>
<td>39.39</td>
<td>40.74</td>
<td>20.95</td>
<td>20.93</td>
<td>40.74</td>
</tr>
</tbody>
</table>

Number of Attributes = 5
Number of Descriptors Per Attribute = 20

<table>
<thead>
<tr>
<th>Strategy</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Back-ends</td>
<td>2</td>
<td>32.74</td>
<td>32.74</td>
<td>38.07</td>
<td>40.74</td>
<td>30.84</td>
<td>30.82</td>
<td>40.74</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>32.74</td>
<td>32.74</td>
<td>39.40</td>
<td>40.74</td>
<td>20.95</td>
<td>20.93</td>
<td>40.74</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>32.74</td>
<td>32.74</td>
<td>39.39</td>
<td>40.74</td>
<td>20.95</td>
<td>20.93</td>
<td>40.74</td>
</tr>
</tbody>
</table>

Number of Attributes = 5
Number of Descriptors Per Attribute = 30

<table>
<thead>
<tr>
<th>Strategy</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Back-ends</td>
<td>2</td>
<td>32.74</td>
<td>32.74</td>
<td>38.07</td>
<td>40.74</td>
<td>30.84</td>
<td>30.82</td>
<td>40.74</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>32.74</td>
<td>32.74</td>
<td>39.40</td>
<td>40.74</td>
<td>20.95</td>
<td>20.93</td>
<td>40.74</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>32.74</td>
<td>32.74</td>
<td>39.39</td>
<td>40.74</td>
<td>20.95</td>
<td>20.93</td>
<td>40.74</td>
</tr>
</tbody>
</table>

Note: There are two rows corresponding to each value of the number of back-ends. The first row gives the average-case times and the second gives the worst-case times.

Figure 16. Directory-Processing-and-Message-Exchanging Times (in msecs) Under Different Strategies
<table>
<thead>
<tr>
<th>Strategy</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32.74</td>
<td>32.74</td>
<td>38.07</td>
<td>40.74</td>
<td>30.84</td>
<td>30.82</td>
<td>40.74</td>
<td>32.74</td>
</tr>
<tr>
<td>2</td>
<td>32.82</td>
<td>32.82</td>
<td>38.15</td>
<td>40.82</td>
<td>30.89</td>
<td>40.74</td>
<td>40.82</td>
<td>32.82</td>
</tr>
<tr>
<td></td>
<td>32.74</td>
<td>32.74</td>
<td>39.40</td>
<td>40.74</td>
<td>20.95</td>
<td>20.93</td>
<td>40.74</td>
<td>32.74</td>
</tr>
<tr>
<td>5</td>
<td>32.82</td>
<td>32.82</td>
<td>39.48</td>
<td>40.82</td>
<td>20.96</td>
<td>40.66</td>
<td>40.82</td>
<td>32.82</td>
</tr>
<tr>
<td></td>
<td>32.74</td>
<td>32.74</td>
<td>39.85</td>
<td>40.74</td>
<td>20.95</td>
<td>20.93</td>
<td>40.74</td>
<td>32.74</td>
</tr>
<tr>
<td>8</td>
<td>32.82</td>
<td>32.82</td>
<td>39.93</td>
<td>40.82</td>
<td>20.96</td>
<td>40.66</td>
<td>40.82</td>
<td>32.82</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Strategy</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32.82</td>
<td>32.82</td>
<td>38.15</td>
<td>40.82</td>
<td>30.89</td>
<td>30.84</td>
<td>40.82</td>
<td>32.82</td>
</tr>
<tr>
<td></td>
<td>33.02</td>
<td>33.02</td>
<td>38.35</td>
<td>41.02</td>
<td>31.01</td>
<td>40.82</td>
<td>41.02</td>
<td>33.02</td>
</tr>
<tr>
<td>5</td>
<td>32.82</td>
<td>32.82</td>
<td>39.48</td>
<td>40.82</td>
<td>20.96</td>
<td>20.93</td>
<td>40.82</td>
<td>32.82</td>
</tr>
<tr>
<td></td>
<td>33.02</td>
<td>33.02</td>
<td>39.68</td>
<td>41.02</td>
<td>21.00</td>
<td>40.70</td>
<td>41.02</td>
<td>33.02</td>
</tr>
<tr>
<td>8</td>
<td>32.82</td>
<td>32.82</td>
<td>39.93</td>
<td>40.82</td>
<td>20.96</td>
<td>20.93</td>
<td>40.82</td>
<td>32.82</td>
</tr>
<tr>
<td></td>
<td>33.02</td>
<td>33.02</td>
<td>40.13</td>
<td>41.02</td>
<td>21.00</td>
<td>40.70</td>
<td>41.02</td>
<td>33.02</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Strategy</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32.82</td>
<td>32.82</td>
<td>38.15</td>
<td>40.82</td>
<td>30.89</td>
<td>30.84</td>
<td>40.82</td>
<td>32.82</td>
</tr>
<tr>
<td>2</td>
<td>33.02</td>
<td>33.02</td>
<td>38.35</td>
<td>41.02</td>
<td>31.01</td>
<td>40.82</td>
<td>41.02</td>
<td>33.02</td>
</tr>
<tr>
<td>5</td>
<td>32.82</td>
<td>32.82</td>
<td>39.48</td>
<td>40.82</td>
<td>20.96</td>
<td>20.93</td>
<td>40.82</td>
<td>32.82</td>
</tr>
<tr>
<td>8</td>
<td>32.82</td>
<td>32.82</td>
<td>39.93</td>
<td>40.82</td>
<td>20.96</td>
<td>20.93</td>
<td>40.82</td>
<td>32.82</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Strategy</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32.82</td>
<td>32.82</td>
<td>38.15</td>
<td>40.82</td>
<td>30.89</td>
<td>30.84</td>
<td>40.82</td>
<td>32.82</td>
</tr>
<tr>
<td>2</td>
<td>33.02</td>
<td>33.02</td>
<td>38.35</td>
<td>41.02</td>
<td>31.01</td>
<td>40.82</td>
<td>41.02</td>
<td>33.02</td>
</tr>
<tr>
<td>5</td>
<td>32.82</td>
<td>32.82</td>
<td>39.48</td>
<td>40.82</td>
<td>20.96</td>
<td>20.93</td>
<td>40.82</td>
<td>32.82</td>
</tr>
<tr>
<td>8</td>
<td>32.82</td>
<td>32.82</td>
<td>39.93</td>
<td>40.82</td>
<td>20.96</td>
<td>20.93</td>
<td>40.82</td>
<td>32.82</td>
</tr>
</tbody>
</table>

Number of Attributes = 10
Number of Descriptors Per Attribute = 30

<table>
<thead>
<tr>
<th>Strategy</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32.82</td>
<td>32.82</td>
<td>38.15</td>
<td>40.82</td>
<td>30.89</td>
<td>30.84</td>
<td>40.82</td>
<td>32.82</td>
</tr>
<tr>
<td>2</td>
<td>33.02</td>
<td>33.02</td>
<td>38.35</td>
<td>41.02</td>
<td>31.01</td>
<td>40.82</td>
<td>41.02</td>
<td>33.02</td>
</tr>
<tr>
<td>5</td>
<td>32.82</td>
<td>32.82</td>
<td>39.48</td>
<td>40.82</td>
<td>20.96</td>
<td>20.93</td>
<td>40.82</td>
<td>32.82</td>
</tr>
<tr>
<td>8</td>
<td>32.82</td>
<td>32.82</td>
<td>39.93</td>
<td>40.82</td>
<td>20.96</td>
<td>20.93</td>
<td>40.82</td>
<td>32.82</td>
</tr>
</tbody>
</table>

Figure 16. (Contd.)
<table>
<thead>
<tr>
<th>Strategy Back-ends</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>32.82</td>
<td>32.82</td>
<td>38.15</td>
<td>40.82</td>
<td>30.89</td>
<td>30.84</td>
<td>40.82</td>
<td>32.82</td>
</tr>
<tr>
<td>5</td>
<td>32.82</td>
<td>32.82</td>
<td>38.35</td>
<td>41.02</td>
<td>31.01</td>
<td>40.82</td>
<td>41.02</td>
<td>33.02</td>
</tr>
<tr>
<td>8</td>
<td>32.82</td>
<td>32.82</td>
<td>39.48</td>
<td>40.82</td>
<td>20.96</td>
<td>20.93</td>
<td>40.82</td>
<td>32.82</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Strategy Back-ends</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>32.94</td>
<td>32.94</td>
<td>38.27</td>
<td>40.94</td>
<td>30.96</td>
<td>30.87</td>
<td>40.94</td>
<td>32.94</td>
</tr>
<tr>
<td>5</td>
<td>32.94</td>
<td>32.94</td>
<td>38.55</td>
<td>41.22</td>
<td>31.13</td>
<td>40.94</td>
<td>41.22</td>
<td>33.22</td>
</tr>
<tr>
<td>8</td>
<td>32.94</td>
<td>32.94</td>
<td>39.60</td>
<td>40.94</td>
<td>20.99</td>
<td>20.94</td>
<td>40.94</td>
<td>32.94</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Strategy Back-ends</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>32.94</td>
<td>32.94</td>
<td>38.27</td>
<td>40.94</td>
<td>30.96</td>
<td>30.87</td>
<td>40.94</td>
<td>32.94</td>
</tr>
<tr>
<td>5</td>
<td>32.94</td>
<td>32.94</td>
<td>38.55</td>
<td>41.22</td>
<td>31.13</td>
<td>40.94</td>
<td>41.22</td>
<td>33.22</td>
</tr>
<tr>
<td>8</td>
<td>32.94</td>
<td>32.94</td>
<td>39.60</td>
<td>40.94</td>
<td>20.99</td>
<td>20.94</td>
<td>40.94</td>
<td>32.94</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Strategy Back-ends</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>32.94</td>
<td>32.94</td>
<td>38.27</td>
<td>40.94</td>
<td>30.96</td>
<td>30.87</td>
<td>40.94</td>
<td>32.94</td>
</tr>
<tr>
<td>5</td>
<td>32.94</td>
<td>32.94</td>
<td>38.55</td>
<td>41.22</td>
<td>31.13</td>
<td>40.94</td>
<td>41.22</td>
<td>33.22</td>
</tr>
<tr>
<td>8</td>
<td>32.94</td>
<td>32.94</td>
<td>39.60</td>
<td>40.94</td>
<td>20.99</td>
<td>20.94</td>
<td>40.94</td>
<td>32.94</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Strategy Back-ends</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>32.94</td>
<td>32.94</td>
<td>38.27</td>
<td>40.94</td>
<td>30.96</td>
<td>30.87</td>
<td>40.94</td>
<td>32.94</td>
</tr>
<tr>
<td>5</td>
<td>32.94</td>
<td>32.94</td>
<td>38.55</td>
<td>41.22</td>
<td>31.13</td>
<td>40.94</td>
<td>41.22</td>
<td>33.22</td>
</tr>
<tr>
<td>8</td>
<td>32.94</td>
<td>32.94</td>
<td>39.60</td>
<td>40.94</td>
<td>20.99</td>
<td>20.94</td>
<td>40.94</td>
<td>32.94</td>
</tr>
</tbody>
</table>

Figure 16. (Contd.)
<table>
<thead>
<tr>
<th>Strategy</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>32.94</td>
<td>32.94</td>
<td>38.27</td>
<td>40.94</td>
<td>30.96</td>
<td>30.87</td>
<td>40.94</td>
<td>32.94</td>
</tr>
<tr>
<td></td>
<td>33.22</td>
<td>33.22</td>
<td>38.55</td>
<td>41.22</td>
<td>31.13</td>
<td>40.94</td>
<td>41.22</td>
<td>33.22</td>
</tr>
<tr>
<td>5</td>
<td>32.94</td>
<td>32.94</td>
<td>39.60</td>
<td>40.94</td>
<td>20.99</td>
<td>20.94</td>
<td>40.94</td>
<td>32.94</td>
</tr>
<tr>
<td></td>
<td>33.22</td>
<td>33.22</td>
<td>39.88</td>
<td>41.22</td>
<td>21.04</td>
<td>40.74</td>
<td>41.22</td>
<td>33.22</td>
</tr>
<tr>
<td>8</td>
<td>32.94</td>
<td>32.94</td>
<td>40.05</td>
<td>40.94</td>
<td>20.99</td>
<td>20.93</td>
<td>40.94</td>
<td>32.94</td>
</tr>
<tr>
<td></td>
<td>33.22</td>
<td>33.22</td>
<td>40.33</td>
<td>41.22</td>
<td>21.04</td>
<td>40.70</td>
<td>41.22</td>
<td>33.22</td>
</tr>
</tbody>
</table>

Figure 16. (Contd.)
Thus, all the descriptor processing is performed at this single back-end and parallel descriptor processing is not achieved. In Strategy E, on the other hand, the duplication of descriptors allows us to achieve parallel descriptor processing for all types of query conjunctions.

In addition, the following observations may be made from the results of Figure 16. Under Strategies E and F, the performance of MDBS improves with an increase in the number of back-ends. However, this improvement will not go further if the number of back-ends is greater than the number of predicates in a query conjunction. The performance of MDBS under other strategies, does not improve with increasing number of back-ends. Thus, these other strategies are not suitable for implementation in MDBS.

The results of Figure 16 do not clearly indicate whether the performance of MDBS under any strategy is independent of the number of descriptors per directory attribute. To test whether the performance of MDBS is independent of the number of descriptors per attribute, we make a second set of calculations with the number of descriptors per attribute varying between 100 and 200. The results are shown in Figure 17. Comparing corresponding entries in Figure 16 with the ones in Figure 17, we see that the increasing number of descriptors per attribute affects
### # of Descriptors Per Attribute = 100

<table>
<thead>
<tr>
<th>n</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>32.74</td>
<td>32.74</td>
<td>38.07</td>
<td>40.74</td>
<td>30.84</td>
<td>30.82</td>
<td>40.74</td>
<td>32.74</td>
</tr>
<tr>
<td></td>
<td>56.57</td>
<td>56.57</td>
<td>61.90</td>
<td>64.57</td>
<td>45.14</td>
<td>64.49</td>
<td>40.82</td>
<td>56.57</td>
</tr>
<tr>
<td>5</td>
<td>32.74</td>
<td>32.74</td>
<td>39.40</td>
<td>40.74</td>
<td>20.95</td>
<td>20.93</td>
<td>40.74</td>
<td>32.74</td>
</tr>
<tr>
<td></td>
<td>56.57</td>
<td>56.57</td>
<td>63.23</td>
<td>64.57</td>
<td>25.71</td>
<td>64.41</td>
<td>40.82</td>
<td>56.57</td>
</tr>
<tr>
<td>8</td>
<td>32.74</td>
<td>32.74</td>
<td>38.85</td>
<td>40.74</td>
<td>20.95</td>
<td>20.93</td>
<td>40.74</td>
<td>32.74</td>
</tr>
<tr>
<td></td>
<td>56.57</td>
<td>56.57</td>
<td>63.68</td>
<td>64.57</td>
<td>25.71</td>
<td>64.41</td>
<td>40.82</td>
<td>56.57</td>
</tr>
</tbody>
</table>

### # of Attributes = 5

<table>
<thead>
<tr>
<th>n</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>56.49</td>
<td>56.49</td>
<td>61.82</td>
<td>64.49</td>
<td>45.09</td>
<td>45.07</td>
<td>40.74</td>
<td>56.49</td>
</tr>
<tr>
<td></td>
<td>80.32</td>
<td>80.32</td>
<td>85.65</td>
<td>88.32</td>
<td>59.39</td>
<td>88.24</td>
<td>40.82</td>
<td>80.32</td>
</tr>
<tr>
<td>5</td>
<td>56.49</td>
<td>56.49</td>
<td>63.15</td>
<td>64.49</td>
<td>25.70</td>
<td>25.68</td>
<td>40.74</td>
<td>56.49</td>
</tr>
<tr>
<td></td>
<td>80.32</td>
<td>80.32</td>
<td>86.98</td>
<td>88.32</td>
<td>30.46</td>
<td>88.16</td>
<td>40.82</td>
<td>80.32</td>
</tr>
<tr>
<td>8</td>
<td>56.49</td>
<td>56.49</td>
<td>63.60</td>
<td>64.49</td>
<td>25.70</td>
<td>25.68</td>
<td>40.74</td>
<td>56.49</td>
</tr>
<tr>
<td></td>
<td>80.32</td>
<td>80.32</td>
<td>87.43</td>
<td>88.32</td>
<td>30.46</td>
<td>88.16</td>
<td>40.82</td>
<td>80.32</td>
</tr>
</tbody>
</table>

### # of Attributes = 5

<table>
<thead>
<tr>
<th>n</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>56.49</td>
<td>56.49</td>
<td>61.82</td>
<td>64.49</td>
<td>45.09</td>
<td>45.07</td>
<td>40.74</td>
<td>56.49</td>
</tr>
<tr>
<td></td>
<td>104.07</td>
<td>104.07</td>
<td>109.40</td>
<td>112.07</td>
<td>73.64</td>
<td>111.99</td>
<td>40.82</td>
<td>104.07</td>
</tr>
<tr>
<td>5</td>
<td>56.49</td>
<td>56.49</td>
<td>63.15</td>
<td>64.49</td>
<td>25.70</td>
<td>25.68</td>
<td>40.74</td>
<td>56.49</td>
</tr>
<tr>
<td></td>
<td>104.07</td>
<td>104.07</td>
<td>110.73</td>
<td>112.07</td>
<td>35.21</td>
<td>111.91</td>
<td>40.82</td>
<td>104.07</td>
</tr>
<tr>
<td>8</td>
<td>56.49</td>
<td>56.49</td>
<td>63.60</td>
<td>64.49</td>
<td>25.70</td>
<td>25.68</td>
<td>40.74</td>
<td>56.49</td>
</tr>
<tr>
<td></td>
<td>104.07</td>
<td>104.07</td>
<td>111.18</td>
<td>112.07</td>
<td>35.21</td>
<td>111.91</td>
<td>40.82</td>
<td>104.07</td>
</tr>
</tbody>
</table>

There are two rows corresponding to each value of the number of backends. The first row gives the average case times and the second row gives the worst case times.

**Figure 17. Descriptors - Processing-and-Message-Exchanging Times (in msecs) for Various Directory Management Strategies.**
<table>
<thead>
<tr>
<th>n</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>32.82</td>
<td>32.82</td>
<td>38.15</td>
<td>40.82</td>
<td>30.89</td>
<td>30.84</td>
<td>40.82</td>
<td>32.82</td>
</tr>
<tr>
<td></td>
<td>56.77</td>
<td>56.77</td>
<td>62.10</td>
<td>64.77</td>
<td>45.26</td>
<td>64.57</td>
<td>41.02</td>
<td>56.77</td>
</tr>
<tr>
<td>5</td>
<td>32.82</td>
<td>32.82</td>
<td>39.48</td>
<td>40.82</td>
<td>20.96</td>
<td>20.93</td>
<td>40.82</td>
<td>32.82</td>
</tr>
<tr>
<td></td>
<td>56.77</td>
<td>56.77</td>
<td>63.43</td>
<td>64.77</td>
<td>25.75</td>
<td>64.45</td>
<td>41.02</td>
<td>56.77</td>
</tr>
<tr>
<td>8</td>
<td>32.82</td>
<td>32.82</td>
<td>39.93</td>
<td>40.82</td>
<td>20.96</td>
<td>20.93</td>
<td>40.82</td>
<td>32.82</td>
</tr>
<tr>
<td></td>
<td>56.77</td>
<td>56.77</td>
<td>63.88</td>
<td>64.77</td>
<td>25.75</td>
<td>64.45</td>
<td>41.02</td>
<td>56.77</td>
</tr>
</tbody>
</table>

# of Attributes = 10
# of Descriptors Per Attribute = 200

<table>
<thead>
<tr>
<th>n</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>56.57</td>
<td>56.57</td>
<td>61.90</td>
<td>64.57</td>
<td>45.14</td>
<td>45.09</td>
<td>40.82</td>
<td>56.57</td>
</tr>
<tr>
<td></td>
<td>80.52</td>
<td>80.52</td>
<td>85.85</td>
<td>88.52</td>
<td>59.51</td>
<td>88.32</td>
<td>41.02</td>
<td>80.52</td>
</tr>
<tr>
<td>5</td>
<td>56.57</td>
<td>56.57</td>
<td>63.23</td>
<td>64.57</td>
<td>25.71</td>
<td>25.68</td>
<td>40.82</td>
<td>56.57</td>
</tr>
<tr>
<td></td>
<td>80.52</td>
<td>80.52</td>
<td>87.18</td>
<td>88.52</td>
<td>30.50</td>
<td>88.20</td>
<td>41.02</td>
<td>80.52</td>
</tr>
<tr>
<td>8</td>
<td>56.57</td>
<td>56.57</td>
<td>64.68</td>
<td>64.57</td>
<td>25.71</td>
<td>25.68</td>
<td>50.82</td>
<td>56.57</td>
</tr>
<tr>
<td></td>
<td>80.52</td>
<td>80.52</td>
<td>87.63</td>
<td>88.52</td>
<td>30.50</td>
<td>88.20</td>
<td>41.02</td>
<td>80.52</td>
</tr>
</tbody>
</table>

# of Attributes = 10
# of Descriptors Per Attribute = 300

<table>
<thead>
<tr>
<th>n</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>56.57</td>
<td>56.57</td>
<td>61.90</td>
<td>64.57</td>
<td>45.14</td>
<td>45.09</td>
<td>40.82</td>
<td>56.57</td>
</tr>
<tr>
<td></td>
<td>104.27</td>
<td>104.27</td>
<td>109.60</td>
<td>112.27</td>
<td>73.76</td>
<td>112.07</td>
<td>41.02</td>
<td>104.27</td>
</tr>
<tr>
<td>5</td>
<td>56.57</td>
<td>56.57</td>
<td>63.23</td>
<td>64.57</td>
<td>25.71</td>
<td>25.68</td>
<td>40.82</td>
<td>56.57</td>
</tr>
<tr>
<td></td>
<td>102.37</td>
<td>104.27</td>
<td>110.93</td>
<td>112.27</td>
<td>35.25</td>
<td>111.95</td>
<td>41.02</td>
<td>104.27</td>
</tr>
<tr>
<td>8</td>
<td>56.57</td>
<td>56.57</td>
<td>63.68</td>
<td>64.57</td>
<td>25.71</td>
<td>25.68</td>
<td>40.82</td>
<td>56.57</td>
</tr>
<tr>
<td></td>
<td>104.27</td>
<td>104.27</td>
<td>111.38</td>
<td>112.27</td>
<td>35.25</td>
<td>111.95</td>
<td>41.02</td>
<td>104.27</td>
</tr>
</tbody>
</table>

# of Attributes = 15
# of Descriptors Per Attribute = 100

<table>
<thead>
<tr>
<th>n</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>32.94</td>
<td>32.94</td>
<td>38.27</td>
<td>40.94</td>
<td>30.96</td>
<td>30.87</td>
<td>40.94</td>
<td>32.94</td>
</tr>
<tr>
<td></td>
<td>56.97</td>
<td>56.97</td>
<td>62.30</td>
<td>64.97</td>
<td>45.38</td>
<td>64.69</td>
<td>41.22</td>
<td>56.97</td>
</tr>
<tr>
<td>5</td>
<td>32.94</td>
<td>32.94</td>
<td>39.60</td>
<td>40.94</td>
<td>20.99</td>
<td>20.94</td>
<td>40.94</td>
<td>32.94</td>
</tr>
<tr>
<td></td>
<td>56.97</td>
<td>56.97</td>
<td>63.63</td>
<td>64.97</td>
<td>25.79</td>
<td>64.49</td>
<td>41.22</td>
<td>56.97</td>
</tr>
<tr>
<td>8</td>
<td>32.94</td>
<td>32.94</td>
<td>40.05</td>
<td>40.94</td>
<td>20.99</td>
<td>20.93</td>
<td>40.94</td>
<td>32.94</td>
</tr>
<tr>
<td></td>
<td>56.97</td>
<td>56.97</td>
<td>64.08</td>
<td>64.97</td>
<td>25.79</td>
<td>64.45</td>
<td>41.22</td>
<td>56.97</td>
</tr>
</tbody>
</table>

Figure 17. (Contd.)
## # of Descriptors Per Attribute = 200

<table>
<thead>
<tr>
<th>n</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>56.69</td>
<td>56.69</td>
<td>62.02</td>
<td>64.69</td>
<td>45.21</td>
<td>45.21</td>
<td>40.94</td>
<td>56.69</td>
</tr>
<tr>
<td>5</td>
<td>80.72</td>
<td>80.72</td>
<td>86.05</td>
<td>88.72</td>
<td>59.63</td>
<td>88.44</td>
<td>41.22</td>
<td>80.72</td>
</tr>
<tr>
<td>8</td>
<td>56.69</td>
<td>56.69</td>
<td>62.02</td>
<td>64.69</td>
<td>45.21</td>
<td>45.21</td>
<td>40.94</td>
<td>56.69</td>
</tr>
</tbody>
</table>

## # of Attributes = 15

## # of Descriptors Per Attribute = 300

<table>
<thead>
<tr>
<th>n</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>56.69</td>
<td>56.69</td>
<td>62.02</td>
<td>64.69</td>
<td>45.21</td>
<td>45.12</td>
<td>40.94</td>
<td>56.69</td>
</tr>
<tr>
<td>5</td>
<td>104.47</td>
<td>104.47</td>
<td>109.80</td>
<td>112.47</td>
<td>73.88</td>
<td>112.19</td>
<td>41.22</td>
<td>104.47</td>
</tr>
<tr>
<td>8</td>
<td>56.69</td>
<td>56.69</td>
<td>62.02</td>
<td>64.69</td>
<td>45.21</td>
<td>45.12</td>
<td>40.94</td>
<td>56.69</td>
</tr>
</tbody>
</table>

Figure 17. (Contd.)
There are two rows for each value of the number of back-ends. The first row gives the average case times and the second row gives the worst case times.

Figure 18. Descriptor-Processing-and-Message-Exchange Times (in msecs) for Various Directory Management Strategies
<table>
<thead>
<tr>
<th># of Descriptors Per Attribute</th>
<th>700</th>
<th>184</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>127.82</td>
<td>127.82</td>
</tr>
<tr>
<td></td>
<td>223.02</td>
<td>223.02</td>
</tr>
<tr>
<td>5</td>
<td>127.82</td>
<td>127.82</td>
</tr>
<tr>
<td></td>
<td>223.02</td>
<td>223.02</td>
</tr>
<tr>
<td>8</td>
<td>127.82</td>
<td>127.82</td>
</tr>
<tr>
<td></td>
<td>223.02</td>
<td>223.02</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># of Attributes</th>
<th>10</th>
<th># of Descriptors Per Attribute</th>
<th>800</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>127.82</td>
<td>127.82</td>
<td>133.15</td>
</tr>
<tr>
<td></td>
<td>246.77</td>
<td>246.77</td>
<td>252.10</td>
</tr>
<tr>
<td>5</td>
<td>127.82</td>
<td>127.82</td>
<td>134.48</td>
</tr>
<tr>
<td></td>
<td>246.77</td>
<td>246.77</td>
<td>253.43</td>
</tr>
<tr>
<td>8</td>
<td>127.82</td>
<td>127.82</td>
<td>134.93</td>
</tr>
<tr>
<td></td>
<td>246.77</td>
<td>246.77</td>
<td>253.88</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># of Attributes</th>
<th>10</th>
<th># of Descriptors Per Attribute</th>
<th>900</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>151.57</td>
<td>151.57</td>
<td>156.90</td>
</tr>
<tr>
<td></td>
<td>270.52</td>
<td>270.52</td>
<td>275.85</td>
</tr>
<tr>
<td>5</td>
<td>151.57</td>
<td>151.57</td>
<td>158.23</td>
</tr>
<tr>
<td></td>
<td>270.52</td>
<td>270.52</td>
<td>277.18</td>
</tr>
<tr>
<td>8</td>
<td>151.57</td>
<td>151.57</td>
<td>158.68</td>
</tr>
<tr>
<td></td>
<td>270.52</td>
<td>270.52</td>
<td>277.63</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># of Attributes</th>
<th>15</th>
<th># of Descriptors Per Attribute</th>
<th>700</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>127.94</td>
<td>127.94</td>
<td>133.27</td>
</tr>
<tr>
<td></td>
<td>223.22</td>
<td>223.22</td>
<td>228.55</td>
</tr>
<tr>
<td>5</td>
<td>127.94</td>
<td>127.94</td>
<td>134.60</td>
</tr>
<tr>
<td></td>
<td>223.22</td>
<td>223.22</td>
<td>229.88</td>
</tr>
<tr>
<td>8</td>
<td>127.94</td>
<td>127.94</td>
<td>135.94</td>
</tr>
<tr>
<td></td>
<td>223.22</td>
<td>223.22</td>
<td>230.33</td>
</tr>
</tbody>
</table>

Figure 18. (Contd.)
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>127.94</td>
<td>127.94</td>
<td>133.27</td>
<td>135.94</td>
<td>87.96</td>
<td>87.87</td>
<td>88.44</td>
<td>127.94</td>
</tr>
<tr>
<td>5</td>
<td>267.97</td>
<td>246.97</td>
<td>252.30</td>
<td>254.97</td>
<td>159.38</td>
<td>254.69</td>
<td>88.72</td>
<td>246.97</td>
</tr>
<tr>
<td></td>
<td>127.94</td>
<td>127.94</td>
<td>134.60</td>
<td>135.94</td>
<td>39.99</td>
<td>39.94</td>
<td>40.94</td>
<td>127.94</td>
</tr>
<tr>
<td></td>
<td>246.97</td>
<td>246.97</td>
<td>253.63</td>
<td>254.97</td>
<td>63.97</td>
<td>254.49</td>
<td>41.22</td>
<td>246.97</td>
</tr>
</tbody>
</table>

# of Attributes = 15
# of Descriptors Per Attribute = 900

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>151.69</td>
<td>151.69</td>
<td>157.02</td>
<td>159.69</td>
<td>102.21</td>
<td>102.21</td>
<td>88.44</td>
<td>151.69</td>
</tr>
<tr>
<td>5</td>
<td>151.69</td>
<td>151.69</td>
<td>158.35</td>
<td>159.69</td>
<td>44.74</td>
<td>44.69</td>
<td>64.69</td>
<td>151.69</td>
</tr>
<tr>
<td>8</td>
<td>151.69</td>
<td>151.69</td>
<td>158.80</td>
<td>159.69</td>
<td>44.74</td>
<td>44.68</td>
<td>40.94</td>
<td>151.69</td>
</tr>
<tr>
<td></td>
<td>270.72</td>
<td>270.72</td>
<td>277.38</td>
<td>278.72</td>
<td>68.54</td>
<td>278.20</td>
<td>41.22</td>
<td>270.72</td>
</tr>
</tbody>
</table>

Figure 18. (Contd.)
the performance considerably. Also, we see that Strategies E and F provide the best average-case results. In fact, differences as large as 30 msecs are now possible between these two strategies and the remaining strategies. Interestingly enough, the performance of Strategy G is now comparable to that of Strategies E and F.

We make a final set of calculations with the number of descriptors per attribute varying from 700 to 900. Of course, this may be an unreasonably large range of values for the number of descriptors per attribute. However, we are interested in the results of this calculation from a point of view where MDBS is heavily loaded and utilized. The results, shown in Figure 18, indicate that Strategy G is now comparable to, and occasionally even better than, Strategies E and F. By and large, these three strategies which employ parallel processing of predicates by multiple back-ends during the descriptor search phase outperform the other strategies.
C. A Preliminary Conclusion Based on the Performance Equations

The results of our study may be summarized as follows. The three strategies namely, strategies E, F and G, which utilize parallel processing of predicates of a query conjunction during the descriptor search phase, may provide better performance than the other five strategies. Furthermore, the employment of any of these strategies in MDBS leads to an improvement in performance with each increase in the number of back-ends. However, the extremely poor worst-case performance of Strategy F would eliminate it from consideration for MDBS implementation. Similarly, the poor average-and-worst case performance of Strategy C for typical values of number of attributes and number of descriptors per attribute would eliminate it from consideration for MDBS implementation. Consequently, the superior strategy for implementation in MDBS is Strategy E.


Our previous evaluation of various strategies on the basis of performance equations has a number of limitations. First, the effects of the directory management strategy on
other aspects of MDBS are not considered. It is possible that some of the strategies may create bottlenecks at some component of MDBS, thus resulting in very poor overall response times. Second, the effects of queueing delays of requests are neglected. Third, the requirement that Strategies E and F lead to improved performance with an increase in the number of back-ends only when the number of back-ends is no greater than the number of predicates in a query conjunction seems unreasonable. We expect that there will be performance improvement even when the number of back-ends far exceeds the number of predicates in a query conjunction. For example, for Strategy E, F or G, if the number of predicates per query conjunction is five, and there are ten back-ends, two query conjunctions can be handled by MDBS with each back-end processing a predicate. An MDBS with only five back-ends would not be able to handle all the predicates concurrently. Thus, MDBS with five back-ends should perform worse than MDBS with ten back-ends, although the results of our previous study would not indicate this observation. Fourth, the interpretation of the worst-case performance of Strategy F may be misleading. In reality, the performance tends to average out and is close to the performance of the average case. However, looking at the results of Figures 16, 17 and 18, one may be tempted to remove Strategy F from consideration for MDBS
implementation. Finally, the strong point of strategies C and D has not been brought out by the study. Their advantage comes from the fact that multiple queries may be simultaneously executed in the different back-ends. That is, they benefit from inter-query parallelism (where several query conjunctions may be processed in parallel by the back-ends) rather than intra-query parallelism (where only predicates of a query conjunction may be processed in parallel by the back-ends).

E. Performance Analysis Based on a Closed Queueing Network Model

We intend to compare the various directory management strategies by using a closed queueing network model. Such a model overcomes all the limitations of the previous study. First of all, we are going to model all system activities in MDBS including parsing of requests, descriptor processing, address generation, retrieval of records from the secondary storage, messages passing, and record processing in the back-ends. Even the bus in MDBS will be modelled. Furthermore, such a model takes into account the queueing delays at each point in the system and also accounts for both inter-query and intra-query parallelisms. The model is shown in Figure 19. It consists of two subsystems: the
MDBS subsystem (which consists of the controller, the back-ends, the disk drives and the bus) and the terminal subsystem. Each terminal is manned by a user who alternates between thinking and waiting. In the thinking state, the user is contemplating what request to submit next. On submitting a new request to MDBS, the user enters the waiting state, where he will remain until MDBS completes the request execution. The mean time that a user spends in a thinking state is called the think time and is denoted \( Z \). The mean time that a user spends in a waiting state is the response time of MDBS and is denoted by \( R \). Thus, the mean time a user spends at a terminal is \( Z+R \), since the user is either thinking or waiting at the terminal. Furthermore, \( R/(Z+R) \) represents the portion of the time that a user is expected to be in the waiting state. Since there are \( M \) users, the number of users who are expected to be in the waiting state is therefore \( MR/(Z+R) \).

We note that the time spent in the waiting state is equivalent to the response time required of MDBS. The number of users in the waiting state is therefore equivalent to the number of responses (i.e., requests to be processed in MDBS). From now on, we say the number of user requests in MDBS whenever we mean the number of users who are in the waiting state. Consequently, we consider the MDBS subsystem alone as a closed queueing network model where the number of
Figure 19. Closed Queueing Network Model of MDBS
requests in the subsystem is MR/(Z+R).

Let us now describe the closed queueing network model of MDBS that is used in our study. The various components in a closed queueing network model are usually referred to as devices. Our model of MDBS has 2(n+1) devices. These are the controller, the bus, the n disk systems and the n back-ends.

A separate I/O submodel is used for the disk system. This I/O submodel is an integral part of the overall model and will be discussed in great detail in the following section. The I/O submodel will be used to calculate the response time of the disk system to an I/O request of a track of data.

E.1 The I/O Submodel for Single Requests

We consider the disk system as consisting of M disk drives and a single channel. The I/O submodel consists of M drive queues and one channel queue as shown in Figure 20. It is assumed that requests are independent and arrive randomly in time at the drive queues, each disk drive being likely addressed equally. The inter-arrival distribution is chosen as exponential with L as the mean rate at which requests are received by the disk system. The inter-arrival
Figure 20. Queueing Model of a Single Channel Disk System
distribution of requests to each drive queue is also exponential with a mean request rate of $L/M$. The assumption of exponential distribution is made because it is the most reasonable one in the absence of other information about the inter-arrival distribution.

For a queue there is a certain rate at which requests will arrive to the queue and a certain rate at which requests in the queue are serviced. It is well-known that a queue may be completely analyzed if we know the mean and variance of the inter-arrival time of requests to the queue and the mean and variance of the service time of requests in the queue. Let us now analyze the drive queues and the channel queue, in turn.

Analyzing a Drive Queue - The drive queue service time is the seek time. The seek time of a disk drive is approximated by an equation of the form $a+by$, where $a$ and $b$ are constants and $y$ is the number of cylinders traversed during the seek. We let $N$ be the total number of cylinders in a disk drive. Then,

Mean service time = Mean seek time

\[ \text{Mean service time} = \text{sig}(1,n)(a+by)(2/N - 2y/N^2) \]

\[ = a - a/N + bN/3 - b/3N. \]

Variance of service time = Variance of seek time

\[ \text{Variance of service time} = (b^2)(N^2)/18 + b^2/18 - a^2/N^2 \]

\[ - (b^2/9)(N^2) - 2ab/3(N^2) + a^2/N. \]
(See Appendix E for the derivation).

For a drive queue, these are the only two quantities we will need in our derivation.

Analyzing the Channel Queue - For the channel queue, the arrival distribution is exponential and the service distribution is constant. Then

\[ \text{Mean service time} = \text{disk rotation time} = d. \]

Variance of service time = 0.

\[ \text{Mean Inter-arrival time} = 1/L \text{ (See Figure 20).} \]

Variance of Inter-arrival time = \(1/(L^2)\).

The four quantities needed to analyze the channel queue are, thus, completely specified. Since the inter-arrival distribution for the channel queue is exponential and the service distribution for the channel queue is non-exponential, the channel queue is now analyzed as an M/G/1 queue [Kle.75]. Application of a standard queueing theory equation gives us

\[ \text{channel wait time} = L(d^2)/(2(1 - Ld)). \]

Let the time spent by a request in the channel queue and the time spent by a request in getting service after its removal from the channel queue be referred to as the time spent by a
request in the channel. Then

Mean time in channel = L(d^2)/(2(1-Ld)) + d.

Variance of time in channel = (L^2)(d^4)/(4((1-Ld)^2))
+ L(d^3)/(3(1-Ld)).

(See Appendix E for the derivation).

This completes our analysis of the channel queue.

Analyzing the Composite Queue - We shall consider a drive queue and channel queue as a composite queue. This is because after a disk drive completes a seek, it must acquire the channel for transferring the track of data over the channel. Only then may a disk drive begin another seek.

Mean service time of Composite Queue

= s
= Mean seek time + Mean time in channel
= a - a/N + bN/3 - b/3N
+ L(d^2)/(2(1-Ld)) + d.

Variance of service time of Composite Queue

= v
= Variance of seek time
+ Variance of time in channel
= (b^2)(N^2)/18 + b^2/18 - a^2/N^2
- b^2/(9(N^2)) - 2ab/(3(N^2)) + a^2/N
+ 2ab/3 + (L^2)(d^4)/(4((1-Ld)^2))
+ L(d^3)/(3(1-Ld)).

Mean Inter-arrival time to composite queue
\[ \text{Variance of inter-arrival time to composite queue} = \frac{M^2}{L} \]

Thus, all the four quantities needed to specify the composite queue are known. Since the inter-arrival distribution of the queue is exponential and the service distribution is non-exponential, the composite queue is an M/G/1 queue. Then, using a standard queueing theory equation, the total time spent by a request in the composite queue which is the total time spent by a request in the disk system and, hence, the response time for a request is \( r \) and

\[ r = \frac{(L/M)(s^2 + \nu)}{2(1 - (Ls/M))} + s \]

**A Note on the Derivation**

The derivation here follows the work of [Gotl73] to a large extent. For instance, the idea of treating the disk module queue and the channel queue as a composite queue is obtained from there. However, there are a number of differences between our work and that of [Gotl73].

The channel queue is modelled as an M/G/1 queue unlike in [Gotl73]. That is, we simplify the analysis by assuming that the requests arrive at the channel queue with exponentially distributed interarrival times. Such an assumption has been made before [Fran74]. This allows us to
get a closed-form expression for the overall response time. Secondly, our channel service time is always taken as the disk rotation time. This is because, in our system, all requests require retrieval of entire tracks at a time. This is also different from [Gotl73]. Finally, the analysis of the average seek time is also different from [Gotl73].

A Note on this model vs. other models - The I/O submodel that we have developed thus far is simpler than other models of disk systems in that we are able to obtain a simple closed-form solution for the response time without relying on iteration and transformation. The results of [Bard81] require the solution of a set of simultaneous equations by the Newton-Raphson method. The method of [Gotl73] for multiple disk drives also requires an iteration of equations to be performed. The use of a machine-repair queueing model [Saat61] for the analysis of the channel queue also yields no closed form solution for the response time. Finally, the results of [Fran74] require Laplace transformations.

E.2 The I/O Submodel for Bulk Requests with Fixed Bulk Size

We now proceed to make a very important extension to the I/O submodel. In a database management system environment such as MDBS where requests are issued in a
high-level query language, each request may involve several tracks. Thus, each request to the disk system requires actually the retrieval of several tracks. In the sequel, we assume that requests to the disk system arrive with an exponential inter-arrival distribution at a mean rate $L$. However, each request is assumed to require the retrieval of $T$ tracks of data which are randomly distributed on the disk tracks. This is a bulk arrival system in the queueing theory terminology. We term $T$ the size of a bulk request. Sometimes we may also talk of the $T$ subrequests of a bulk request.

The queueing model we use is shown in Figure 2. As in the previous section for single requests, we will consider a drive queue and the channel queue as a composite queue. The composite queue is a queue with bulk arrivals. In order to analyze a queue with bulk arrivals, we need to know the mean and variance of the service time, the mean and variance of the inter-arrival time and the first and second moments of the size of a bulk request.

The mean and variance of the service time of the composite queue, $s$ and $v$, are calculated in the previous section. The only difference is that $L$ in the expressions for $s$ and $v$ must now be replaced by $TL$. 
Figure 21. Queueing Model of a Disk System with Bulk Arrivals at Size $T$ at a Rate $L$

Mean = $\frac{T}{M}$

2nd Moment = $\frac{T}{N} \left( 1 - \frac{1}{M} + \frac{T}{M} \right)$

of bulk size
The mean and variance of the inter-arrival time of the composite queue are $1/L$ and $1/L^2$, respectively.

The first moment of the size of a bulk request is $T/M$. The second moment of the size of a bulk request at each module is calculated by assuming a Bernoulli trial as follows. There are $T$ subrequests in a bulk request received by the disk system. For each of these $T$ subrequests, there is a $1/M$ probability that it will be assigned to a particular drive. Then, the probability that $i$ requests out of $T$ will be assigned to the same module is:

$$\text{comb}(T,i)((1/M)^i)(1-(1/M))^{(T-i)}$$

Here, $\text{comb}(a,b)$ stands for the number of subsets of size $n$ that may be formed from the elements of a set of size $M$. Thus, the second moment of the bulk size of a request at a module is

$$\text{sig}(i,1,T)((i^2)\text{comb}(T,i)((1/M)^i)(1-(1/M))^{(T-i)})$$

In the above expression, $\text{sig}(a,b,c)$ stands for the summation over the variable $a$, between the limits of $b$ and $c$, of the expression following in parenthesis. This is simplified to

$$(T/M)(1-L/M + T/M).$$

All the six quantities needed to analyze the composite queue are now known. The total response time may now be calculated using the following formula from [Saat61].

$$r = s/(2(1-q))(t/r' + v/s^2) + s$$
where
\[ q = L * s * r', \]
\[ r' = T / M, \]
\[ t = (T / M)(1 - 1 / M + T / M). \]
Replacing \( T \) with 1, we get the same result that we got for the previous analysis when bulk requests were not assumed. Hence, we have reasons to believe that our analysis is correct.

E.3 The I/O Submodel for Bulk Requests with Variable Bulk Size

We now make one final variation in our assumptions for modelling the I/O system of MDBS. In the above analysis, we assumed that the size of a bulk request, as issued to the disks of back-end, was fixed at \( T \). More generally, a request to MDBS will require the retrieval of \( T \) tracks. However, the number of \( T \) tracks that will be retrieved at any one back-end varies from 1 to \( T \). Therefore, we need to assume that each disk system (one disk system is associated with each back-end) may receive a variable number of subrequests with each bulk request. That is, the size of the bulk requested will vary from 1 to \( T \). If there are \( n \) back-ends, then the mean size of the bulk requested at each disk system is \( T / n \) and the variance of this bulk size is
This explains the queueing model shown in Figure 22. The mean and variance of the bulk size of each request as received by each module must now be calculated. This may be calculated as follows.

The probability that there are \( j \) subrequests of the bulk request at a drive is equal to the probability that \( i \) out of \( T \) requests are first chosen to be sent to one of the back-ends multiplied by the probability that \( j \) out of these \( i \) requests are sent to a particular drive. Then, the mean bulk size at a drive is:

\[
\text{mean} = \text{sig}(i,0,T)(\text{comb}(T,i)(1/n)^i \times (1 - 1/n)^{(T-1)}) \\
\times \text{sig}(j,0,i)(j\text{comb}(i,j)(1/M)^j \times (1 - 1/M)^{(i-j)})
\]

\[
= \frac{T}{Mn}
\]

Similarly, the second moment of bulk size at a drive is:

\[
\text{variance} = \text{sig}(i,0,T)(\text{comb}(T,i)(1/n)^i \times (1 - 1/n)^{(T-1)}) \\
\times \text{sig}(j,0,i)(j^2\text{comb}(i,j)(1/M)^j \times (1 - 1/M)^{(i-j)})
\]

\[
= \left(\frac{T}{Mn}\right)(1 - 1/Mn) + T^2/((M^2)(n^2))
\]
Mean bulk size = \( \frac{T}{n} \)

Variance of bulk size = \( \frac{T}{n} (1 - \frac{1}{n}) \)

Mean bulk size at each module

= \( \frac{T}{Mn} \)

2nd Moment of bulk size at each module

= \( \frac{T}{Mn} (1 - \frac{1}{Mn}) + \frac{Tn^2}{M^2 n^2} \)

Figure 22. Queueing Model of a Disk System With Bulk Arrivals of Variable Size
The final result we are after may now be obtained by starting with the equation developed for bulk requests with fixed bulk size. We let \( r' = T/Mn \), \( t = (T/Mn)(1 - 1/Mn) + (T^2)/(M^2)(n^2) \) and \( L \) in the expressions for \( s \) and \( v \) be replaced by \( LT/n \). This is the result we shall use for the response time of the disk system. Thus,

\[
  r = q(s/(2(1-q))(t/r' + v/s^2) + s \quad \text{where} \quad q = Lsr', \quad r' = T/Mn, \quad t = (T/Mn)(1 - 1/Mn) + (T^2)/(M^2)(n^2)
\]

F. Modelling the Eight Strategies for Evaluation

Having described the I/O submodel thoroughly, we are now ready to describe the overall closed queueing network model of which the I/O submodel is a part. We will develop a separate closed queueing network model for each of the eight strategies. For ease in describing these models, we use the following terminology.

- \( t_{\text{parse}} \): time to parse a user request.
- \( t_{\text{dir}} \): time to do descriptor processing in Strategies A, B, C, D and H.
- \( t_{\text{dire}} \): time to do descriptor processing in Strategy E.
\texttt{tdirf} : time to do descriptor processing in Strategy F.

\texttt{tidrg} : time to do descriptor processing in Strategy G.

\texttt{tm} : time to generate a message.

\texttt{adgen} : time for address generation in the controller.

\texttt{adgenl} : time for address generation in the back-ends.

\texttt{T} : number of tracks to be retrieved for a typical request.

\texttt{y} : average number of predicates in a user query.

\texttt{tbus} : time to send a user request over the bus.

\texttt{tbus2} : time to send retrieved records over bus.

\texttt{tou} : time taken by controller to output retrieved records to user.

\texttt{tproc} : time taken by a back-end to check a track of records against a user query.

We are now ready to describe the eight models for the eight different strategies.
F.1 The Centralized Model

Consider the sequence of execution of a typical request. The request is first processed at the controller. The controller must parse the request and this will take $t_{parse}$ time units. Next, descriptor processing must be performed on the query in the request. We have already shown how the time for descriptor processing ($t_{dir}$) may be calculated for the various strategies. Next, the controller must generate the necessary secondary memory addresses using the augmented CDT. This will take $adgen$ time units. Finally, the controller must broadcast the addresses to the back-ends which will take $tm$ time units. Thus, the total time taken to service the request at the controller is $(tm + t_{dir} + adgen + t_{parse})$. Since each back-end must receive a copy of the request, the controller effectively serves $n$ requests, where the service time of a request is $(tm + t_{dir} + t_{parse} + adgen)/n$.

The request is now sent over the bus to the back-ends. The bus service time for request processing is $t_{bus}$. Since $n$ copies of the same request are sent to $n$ respective back-ends, the bus effectively serves $n$ requests. Thus, the service time is really $t_{bus}/n$. The request is now received by the back-ends which use the respective disk systems to retrieve relevant tracks. If the average number of tracks
to be retrieved for a request is $T$, the average number of tracks that has to be retrieved at each back-end is $T/n$. The service time of a disk system in responding to these $T/n$ I/O requests is obtained from the I/O submodel described in the previous section.

The retrieved tracks are now being checked against the user's query. It is assumed that $t_{proc}$ msecs have to be spent by a back-end in processing a track of records against a user's query. The records satisfying the query are now sent over the bus to the controller. The service time of the bus in this case is assumed to be $t_{bus2}$ (to be $t_{bus2}/n$ when there are $n$ requests). Finally, the controller must output the records to the user which takes $t_{out}$ time units. A good approximation of $t_{out}$ is $t_m$, the time to send a message. As before, it was convenient to assume that actually $n$ separate results are sent to the user with an average service time of $t_{out}/n$.

In some cases, some of the devices (i.e., components of MDBS) have more than one queue. For instance, the controller has two queues. The first queue contains requests that must be parsed, processed, and so on. The second queue contains a group of records waiting to be output to the user. Such cases are handled as separate classes in our model. Thus, there are two classes of
quests at the controller. Let $S(a, b)$ be the service time for the class b request at device a. Similarly let $V(a, b)$ be the visit ratio of the class b request at device a which is defined as the number of service completions of class b requests at device a for each service completion from MDBS. A closed queueing network is completely specified when $V(a, b)$ and $S(a, b)$ are specified for all classes and for all devices. Then, the results of [Rood79] may be used to calculate the response times, average queue lengths, and utilizations on a per class and per device basis. Let us specify $V(a, b)$ and $S(a, b)$ for our model. Let the controller be device 1, the bus be device 2, a disk system be device 3 and a back-end be device 4.

There are two classes of requests at the controller (device 1). The queue of requests waiting to be parsed, processed, and so on, is designated as class 1 at device 1. The queue of records waiting to be output by the controller to the user is designated as class 2 at device 1. The bus (device 2) also has two classes of requests. The queue of messages from the controller to the back-ends is designated as class 1 at device 2. The queue of records sent from the back-ends to the controller to be output to the user is designated as class 2 at device 2. The disk system (device 3) has only one request class. This is the queue of I/O requests for retrieval of tracks of records at specified
addresses which is designated as class 1 at device 3. Finally, each back-end (device 4) has only one request class and this is the queue of records retrieved by the disk system which are waiting to be checked against the user's query. This queue is designated as class 1 at device 4. In some of the later models, we will see that a back-end may have two more request classes, making a total of up to three request classes at a back-end. If a model has two or more request classes at a back-end, the following notation is employed. The queue of requests waiting for address generation is designated as class 1 at device 4. The queue of records retrieved by the disk system which are waiting to be checked against the user's query is designated as class 2 at device 4. Finally, the queue of requests waiting for descriptor processing is designated as class 3 at device 4. This completes our discussion of the notation which will be followed for this model and for the models for all the other strategies also. The S and V matrices for the centralized model is as below.

\[
S(1,1) = \frac{(tdir + tm + tparse + adgen)}{n}
\]

\[
S(1,2) = \frac{tout}{n}
\]

\[
S(2,1) = \frac{tbus}{n}
\]

\[
S(2,2) = \frac{tbus2}{n}
\]

\[
S(3,1) = ?
\]
\[ S(4,1) = t_{\text{proc}} \]
\[ V(1,1) = n \]
\[ V(1,2) = n \]
\[ V(2,1) = n \]
\[ V(2,2) = n \]
\[ V(3,1) = T/n \]
\[ V(4,1) = T/n \]

Note that we have put a question mark for \( S(3,1) \), the service time of the disk system. The I/O submodel will be used to calculate \( S(3,1) \).

F.2 The Partially Centralized Model

The notation we developed for the previous strategy will now be used to explain the remaining models. Let us consider the sequence of execution of a particular request in the partially centralized strategy. First, the controller parses the request and then performs descriptor processing on the request. Finally, it broadcasts the corresponding descriptors to all the back-ends. Thus, the service time for the request at the controller is \((t_{\text{dir}} + t_{\text{parse}} + t_m)/n\). The request now goes over the bus with a service time of \( t_{\text{bus}}/n \) to the back-ends.
The back-ends perform address generation taking adgenl units of time. We use adgenl rather than adgen as in the previous strategy because the address generation in this strategy and in all the remaining strategies is different from the address generation in the centralized strategy.

Next, the request is presented by a back-end to its disk system for accessing the relevant tracks. The retrieved data are then processed in the back-ends taking a service time of tproc units. The results are then sent back over the bus to the controller. The bus service time is again tbus2/n. Finally, the results must be returned to the user and the controller takes tout/n time units for this. The model may be specified by specifying its S and V entries as below.

\[
\begin{align*}
S(1,1) &= \frac{(tdir + tparse + tm)}{n} \\
S(1,2) &= \frac{tout}{n} \\
S(2,1) &= \frac{tbus}{n} \\
S(2,2) &= \frac{tbus2}{n} \\
S(3,1) &= ? \\
S(4,1) &= \text{adgenl} \\
S(4,2) &= \text{tproc} \\
V(1,1) &= n \\
V(1,2) &= n \\
V(2,1) &= n
\end{align*}
\]
\[ V(2,2) = n \]
\[ V(3,1) = \frac{T}{n} \]
\[ V(4,1) = 1 \]
\[ V(4,2) = \frac{T}{n} \]

F.3 The Rotating Model

Consider the sequence of execution of a typical request. Two possibilities exist depending on whether the descriptor processing is done at the controller or at one of the back-ends. If the descriptor processing is done at the controller, then the service time at the controller will be \((t_{\text{parse}} + t_m + t_{\text{dir}})/n\). Otherwise, the service time will be \((t_{\text{parse}} + t_m)/n\). Since there are two possibilities of request execution in this model as compared to the single request execution possibility in the other models, an additional request class is introduced at the controller for this model and this is designated class 3 at device 1. The request is now broadcast over the bus with a service time of \(t_{\text{bus}}/n\). If descriptor processing has not already been done at the controller, then it must be done at the back-ends, and this takes \((t_{\text{dir}} + t_m)\) units. The additional \(t_m\) units is needed to communicate the results to all the back-ends.

Next, the back-ends perform address generation taking \(t_{\text{gen}}\) time units. The disk systems are activated at this time to access tracks. The retrieved data are then processed by the
back-ends with a service time of tproc. Records satisfying the request are sent over the bus taking tbus2/n time units and finally, the controller spends tout/n time units outputting the results to the user. The model may be specified by the following service times and visit ratios.

\[
S(1,1) = (tdir + tparse + tm)/n \\
S(1,2) = tout/n \\
S(1,3) = (tparse + tm)/n \\
S(2,1) = tbus/n \\
S(2,2) = tbus2/n \\
S(3,1) = ? \\
S(4,1) = adgen1 \\
S(4,2) = tproc \\
S(4,3) = (tdir + tm) \\
V(1,1) = n/(n+1) \\
V(1,2) = n \\
V(1,3) = (n**2)/(n+1) \\
V(2,1) = n \\
V(2,2) = n \\
V(3,1) = T/n \\
V(4,1) = 1 \\
V(4,2) = T/n \\
V(4,3) = 1/(n+1)
\]
Consider the execution sequence of a typical request. The controller does parsing on a request before broadcasting it to all the back-ends. The controller service time is \((t_{\text{parse}} + t_m)/n\). The request is now broadcast over the bus with a service time of \(t_{\text{bus}}/n\) and arrives at the back-ends. One of the back-ends will do the descriptor processing for this request. The visit ratio of the descriptor processing queue at each back-end must be adjusted to reflect the fact that only one out of every \(n\) user generated requests will need descriptor processing at that back-end. The back-end that does descriptor processing must then communicate the results to all the back-ends. Hence, the service time of this queue in the back-end is \((t_{\text{dir}} + t_m)\). Then, all the back-ends will perform address generation independently, taking \(t_{\text{adgen}}\) time units. Next, the disk system at a back-end will retrieve \(T/n\) tracks of data. These tracks of data are checked by the back-ends against the user's query taking \(t_{\text{proc}}\) units of time per track. The results are shipped over the bus taking \(t_{\text{bus2}}/n\) time units. Finally, the controller outputs the results to the user taking \(t_{\text{out}}/n\) time units. The model is entirely specified by specifying the service times and visit ratios as below.

\[ S(1,1) = (t_{\text{parse}} + t_m)/n \]
\[ S(1,2) = \frac{\text{t}_{\text{out}}}{n} \]
\[ S(2,1) = \frac{\text{t}_{\text{bus}}}{n} \]
\[ S(2,2) = \frac{\text{t}_{\text{bus2}}}{n} \]
\[ S(3,1) = ? \]
\[ S(4,1) = \text{adgen1} \]
\[ S(4,2) = \text{t}_{\text{proc}} \]
\[ S(4,3) = \text{t}_{\text{dir}} + \text{tm} \]
\[ V(1,1) = n \]
\[ V(1,2) = n \]
\[ V(2,1) = n \]
\[ V(2,2) = n \]
\[ V(3,1) = \frac{T}{n} \]
\[ V(4,1) = 1 \]
\[ V(4,2) = \frac{T}{n} \]
\[ V(4,3) = \frac{1}{n} \]

F 5 The Fully Duplicated Model

This model differs from the previous one only in the way descriptor processing is done. Instead of each back-end doing the descriptor processing for one out of every \( n \) requests, each back-end will do a portion of the descriptor processing for every request. The back-ends must then exchange their results. The service time and visit ratios are shown below for this model:
F 6 The Descriptors Dividing by Attribute Model

The only difference between this and the previous model is that \( t\text{dire} \) is replaced by \( t\text{dirf} \) in the expression for \( S(4,3) \). The value \( t\text{dirf} \) has been calculated in a previous section and is the time for doing descriptor processing on a single predicate by using Strategy F. Each back-end is expected to handle \( y/n \) out of the \( y \) predicates in a user query.
F.7 The Descriptors Division Within Attribute Model

Once again, this model is very similar to that for Strategy E and may be obtained from there by setting $S(4,3)$ equal to $(y \cdot t_{d\text{irg}} + t_m)$. Once again, $t_{d\text{irg}}$ is the time for descriptor processing on a single predicate using Strategy G and has been calculated in an earlier section. Note that, unlike the previous strategies, all of the predicates in a query are handled at each back-end.

F.8 The Fully Replicated Model

Consider the sequence of execution of a typical request. The controller first parses the request and broadcasts it to all the back-ends. Thus, the controller service time is $(t_{\text{parse}} + t_m)/n$. The requests go over the bus with an average service time of $t_{\text{bus}}/n$. The requests are now received at each back-end. Each back-end will first perform descriptor processing and then address generation on a request. Next $T/n$ track retrieve requests are submitted to the disk system. The retrieved records are searched against the user's query in the controller and this takes $t_{\text{proc}}$ time units for each track of records. The qualified records are now returned via the bus taking a service time of $t_{\text{bus}2}/n$ and are then output to the user by the controller.
taking a service time of \( \text{tout}/n \). The complete specification of the service times and visit ratios is shown below:

\[
\begin{align*}
S(1,1) &= (\text{tparse} + \text{tm})/n \\
S(1,2) &= \text{tout}/n \\
S(2,1) &= \text{tbus}/n \\
S(2,2) &= \text{tbus}2/n \\
S(3,1) &= ? \\
S(4,1) &= \text{tdir} + \text{adgen}1 \\
S(4,2) &= \text{tproc} \\
V(1,1) &= n \\
V(1,2) &= n \\
V(2,1) &= n \\
V(2,2) &= n \\
V(3,1) &= T/n \\
V(4,1) &= 1 \\
V(4,2) &= T/n
\end{align*}
\]

This completes our specification of the various models to be employed. The only question to be answered concerns the incorporation of the I/O submodel into the overall models. We shall explain the procedure employed in order to incorporate the I/O submodel into the overall models in the next section.
G. Results of the Queueing Network Modeling of Strategies

In order to solve any of the eight closed queueing network models of MDBS, the service times and visit ratios of all the devices in the network must be specified. We have specified the visit ratios of all the devices including the disk systems. However, the service times of the disk systems have not yet been specified. In this section, we will describe how the service time of a disk system may be calculated.

In Section E.1, we have an expression for the mean service time of the composite queue, in a disk system, in terms of $L$, the arrival rate of requests to the disk system. However, we did not know then the value of $L$. Therefore, a technique for calculating $L$ and, hence, $s$ is developed herein. We use the following iterative procedure. The following values are used for the various parameters in our calculations.

$t_{parse}$ : 20 msecs
$t_{dir}$ : calculated using equation (2) of Section A
$t_{dire}$ : calculated using equation (3) of Section A
$t_{dirf}$ : calculated using equation (4) of Section A
$t_{dirg}$ : calculated using equation (5) of Section A
$t_{m}$ : 8 msecs
The number of back-ends is taken from the set \([3, 6, 9]\) and the number of requests in the system is taken from the set \([5, 10, 15]\).

Assume an initial value for \(L\), the arrival rate of requests to the disk system as follows:

\[
\frac{1}{(\text{seek time} + \text{rotation time})}
\]

Using this value of \(L\), calculate the value of \(s\), the disk system service time and \(r\), the disk system response time using the expressions derived in Sections E.1 and E.3, respectively. With \(s\), the closed queueing network model is completely specified. It may then be used to calculate utilizations, response times, throughputs and average queue lengths on a per class and per device manner. In particular, the response time of the disk system modelled by the closed queueing network can be calculated. The value of the disk system response time obtained from the closed
queueing network model, $R$ is compared to that obtained from the I/O submodel, $r$. If the two are less than one millisecond apart, the iteration stops. Otherwise, the value of $L$ is modified to be equal to the throughput $T_p$ of the disk system calculated from the closed queueing network model. The iteration procedure is now repeated until the small difference between $R$ and $r$ is obtained. In our experiments with this iteration technique, we found that no more than three iterations were needed. That is, with the chosen initial value of $L$, the convergence was extremely fast. With the closed queueing network models for various directory processing strategies being completely specified, we now present the results of our experiments. The results for strategies A to D are presented in Figures 23, 24, 25 and 26, respectively. Since the results for Strategies E and F are identical, they are presented in Figure 27. Finally, the results for Strategies G and H are presented in Figures 28 and 29, respectively. Each of these figures contains two columns, one for the utilization and one for the response time these two measures are recorded on a per-class-and-per-device basis. The order of presentation of the results within a figure for a particular strategy is the same as the order of presentation of the service time and visit ratio in the description of the model for that Strategy. For instance, in the description of the model for
Strategy A, we first presented the service time and visit ratio for class 1 at device 1. Thus, in Figure 23, the first row gives the utilization and response time for class 1 at device 1. In reading these figures, we should keep in mind that device 1 is the controller, device 2 is the bus, device 3 is a disk system and device 4 is a back-end.

We also present a comparative set of results for the eight different strategies in Figure 30. Each entry in this figure represents the overall response time of MDBS using one of these eight strategies. Most of the discussion that follows is based on the results shown in this figure. However, in order to explain some of the results shown in this figure, we will have to refer back to the results presented in Figures 23 through 29.

As expected, the centralized strategy (i.e., Strategy A) is the worst of the lot. It is consistently the worst strategy over the entire range of values for the number of requests and the number of back-ends. There are situations when the response time obtained by use of the centralized strategy is almost 50% higher than when using some of the other strategies. For example, when the number of back-ends is taken as nine and the number of requests is fifteen, the response time of the centralized strategy is 1255.16 msecs. On the other hand, the response time using Strategy E, the
<table>
<thead>
<tr>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.3178</td>
<td>36.499</td>
<td>.5966</td>
<td>26.191</td>
<td>.7815</td>
<td>22.251</td>
</tr>
<tr>
<td>.0336</td>
<td>2.759</td>
<td>.0630</td>
<td>1.419</td>
<td>.0826</td>
<td>0.964</td>
</tr>
<tr>
<td>.0</td>
<td>0.003</td>
<td>.0001</td>
<td>0.002</td>
<td>.0001</td>
<td>0.0001</td>
</tr>
<tr>
<td>.0672</td>
<td>5.714</td>
<td>.1261</td>
<td>3.032</td>
<td>.1652</td>
<td>2.102</td>
</tr>
<tr>
<td>.9966</td>
<td>172.960</td>
<td>.9506</td>
<td>144.067</td>
<td>.8453</td>
<td>119.090</td>
</tr>
<tr>
<td>.0252</td>
<td>1.026</td>
<td>.0236</td>
<td>1.023</td>
<td>.0206</td>
<td>1.020</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.3189</td>
<td>37.047</td>
<td>.6248</td>
<td>32.458</td>
<td>.8647</td>
<td>39.183</td>
</tr>
<tr>
<td>.0337</td>
<td>2.760</td>
<td>.0630</td>
<td>1.427</td>
<td>.0914</td>
<td>0.977</td>
</tr>
<tr>
<td>0.000</td>
<td>0.003</td>
<td>.0001</td>
<td>0.002</td>
<td>.0001</td>
<td>0.001</td>
</tr>
<tr>
<td>.0674</td>
<td>5.719</td>
<td>.1320</td>
<td>3.071</td>
<td>.1827</td>
<td>2.169</td>
</tr>
<tr>
<td>1.000</td>
<td>369.856</td>
<td>.9955</td>
<td>323.830</td>
<td>.9353</td>
<td>238.085</td>
</tr>
<tr>
<td>.0253</td>
<td>1.026</td>
<td>.0248</td>
<td>1.025</td>
<td>.0228</td>
<td>1.023</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.3189</td>
<td>37.051</td>
<td>.6274</td>
<td>33.675</td>
<td>.8924</td>
<td>53.761</td>
</tr>
<tr>
<td>.0337</td>
<td>2.760</td>
<td>.0663</td>
<td>1.428</td>
<td>.0943</td>
<td>0.981</td>
</tr>
<tr>
<td>0.000</td>
<td>0.003</td>
<td>.0001</td>
<td>0.002</td>
<td>.0001</td>
<td>0.001</td>
</tr>
<tr>
<td>.0674</td>
<td>5.719</td>
<td>.1326</td>
<td>3.074</td>
<td>.1886</td>
<td>2.188</td>
</tr>
<tr>
<td>1.000</td>
<td>567.697</td>
<td>.9996</td>
<td>520.885</td>
<td>.9652</td>
<td>370.881</td>
</tr>
<tr>
<td>.0253</td>
<td>1.026</td>
<td>.0249</td>
<td>1.025</td>
<td>.0236</td>
<td>1.024</td>
</tr>
</tbody>
</table>

Figure 23. Queueing Network Model Results for Strategy A
<table>
<thead>
<tr>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.1592</td>
<td>14.973</td>
<td>.3019</td>
<td>8.693</td>
<td>.4192</td>
<td>6.526</td>
</tr>
<tr>
<td>.0336</td>
<td>2.759</td>
<td>.0638</td>
<td>1.419</td>
<td>.0885</td>
<td>0.967</td>
</tr>
<tr>
<td>.0000</td>
<td>0.003</td>
<td>.0001</td>
<td>0.002</td>
<td>.0001</td>
<td>0.001</td>
</tr>
<tr>
<td>.0672</td>
<td>5.714</td>
<td>.1275</td>
<td>3.028</td>
<td>.1771</td>
<td>2.112</td>
</tr>
<tr>
<td>.9978</td>
<td>168.301</td>
<td>.9613</td>
<td>134.016</td>
<td>.9063</td>
<td>117.194</td>
</tr>
<tr>
<td>.1136</td>
<td>30.361</td>
<td>.1469</td>
<td>21.320</td>
<td>.1230</td>
<td>12.507</td>
</tr>
<tr>
<td>.0252</td>
<td>1.026</td>
<td>.0239</td>
<td>1.023</td>
<td>.0221</td>
<td>1.021</td>
</tr>
</tbody>
</table>

- **Number of Requests** = 5
- **Number of Back-ends** = 6

<table>
<thead>
<tr>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.1595</td>
<td>15.023</td>
<td>.3139</td>
<td>9.193</td>
<td>.4612</td>
<td>7.714</td>
</tr>
<tr>
<td>.0337</td>
<td>2.760</td>
<td>.0663</td>
<td>1.428</td>
<td>.0974</td>
<td>0.984</td>
</tr>
<tr>
<td>.0000</td>
<td>0.003</td>
<td>.0001</td>
<td>0.002</td>
<td>.0001</td>
<td>0.001</td>
</tr>
<tr>
<td>.0674</td>
<td>5.719</td>
<td>.1326</td>
<td>3.074</td>
<td>.1948</td>
<td>2.205</td>
</tr>
<tr>
<td>1.0000</td>
<td>355.655</td>
<td>.9997</td>
<td>325.161</td>
<td>.9970</td>
<td>294.535</td>
</tr>
<tr>
<td>.0253</td>
<td>1.026</td>
<td>.0249</td>
<td>1.025</td>
<td>.0244</td>
<td>1.025</td>
</tr>
</tbody>
</table>

- **Number of Requests** = 10
- **Number of Back-ends** = 9

<table>
<thead>
<tr>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.1595</td>
<td>15.023</td>
<td>.3140</td>
<td>9.203</td>
<td>.4625</td>
<td>7.826</td>
</tr>
<tr>
<td>.0337</td>
<td>2.760</td>
<td>.0663</td>
<td>1.428</td>
<td>.0977</td>
<td>0.985</td>
</tr>
<tr>
<td>.0000</td>
<td>0.003</td>
<td>.0001</td>
<td>0.002</td>
<td>.0001</td>
<td>0.001</td>
</tr>
<tr>
<td>.0674</td>
<td>5.719</td>
<td>.1326</td>
<td>3.074</td>
<td>.1954</td>
<td>2.209</td>
</tr>
<tr>
<td>1.0000</td>
<td>563.502</td>
<td>1.0000</td>
<td>526.068</td>
<td>.9999</td>
<td>497.486</td>
</tr>
<tr>
<td>.1136</td>
<td>30.417</td>
<td>.1528</td>
<td>21.749</td>
<td>.1358</td>
<td>12.864</td>
</tr>
<tr>
<td>.0253</td>
<td>1.026</td>
<td>.0249</td>
<td>1.026</td>
<td>.0244</td>
<td>1.025</td>
</tr>
</tbody>
</table>

- **Number of Requests** = 15
- **Number of Back-ends** = 9

Figure 24. Queueing Network Model Results for Strategy B
<table>
<thead>
<tr>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.0398</td>
<td>13.145</td>
<td>.0433</td>
<td>6.582</td>
<td>.0423</td>
<td>4.378</td>
</tr>
<tr>
<td>.0336</td>
<td>2.759</td>
<td>.0640</td>
<td>1.419</td>
<td>.0894</td>
<td>0.967</td>
</tr>
<tr>
<td>.0883</td>
<td>10.226</td>
<td>.1919</td>
<td>5.674</td>
<td>.2816</td>
<td>4.124</td>
</tr>
<tr>
<td>.0000</td>
<td>0.003</td>
<td>.0001</td>
<td>0.002</td>
<td>.0001</td>
<td>0.001</td>
</tr>
<tr>
<td>.0673</td>
<td>5.714</td>
<td>.1279</td>
<td>3.029</td>
<td>.1788</td>
<td>2.115</td>
</tr>
<tr>
<td>.9981</td>
<td>167.974</td>
<td>.9645</td>
<td>133.882</td>
<td>.9152</td>
<td>117.438</td>
</tr>
<tr>
<td>.1134</td>
<td>30.363</td>
<td>.1473</td>
<td>21.358</td>
<td>.1243</td>
<td>12.517</td>
</tr>
<tr>
<td>.0252</td>
<td>1.026</td>
<td>.0240</td>
<td>1.023</td>
<td>.0224</td>
<td>1.021</td>
</tr>
<tr>
<td>.0188</td>
<td>18.218</td>
<td>.0204</td>
<td>18.231</td>
<td>.0200</td>
<td>18.211</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.0399</td>
<td>13.511</td>
<td>.0449</td>
<td>6.609</td>
<td>.0462</td>
<td>4.412</td>
</tr>
<tr>
<td>.0337</td>
<td>2.760</td>
<td>.0663</td>
<td>1.428</td>
<td>.0976</td>
<td>0.985</td>
</tr>
<tr>
<td>.0885</td>
<td>10.239</td>
<td>.1989</td>
<td>5.824</td>
<td>.3074</td>
<td>4.481</td>
</tr>
<tr>
<td>.0000</td>
<td>0.003</td>
<td>.0001</td>
<td>0.002</td>
<td>.0001</td>
<td>0.001</td>
</tr>
<tr>
<td>.0674</td>
<td>5.719</td>
<td>.1326</td>
<td>3.074</td>
<td>.1952</td>
<td>2.207</td>
</tr>
<tr>
<td>1.0000</td>
<td>365.406</td>
<td>.9999</td>
<td>326.398</td>
<td>.9988</td>
<td>300.118</td>
</tr>
<tr>
<td>.0253</td>
<td>1.026</td>
<td>.0249</td>
<td>1.025</td>
<td>.0244</td>
<td>1.025</td>
</tr>
<tr>
<td>.0188</td>
<td>18.222</td>
<td>.0212</td>
<td>18.265</td>
<td>.0218</td>
<td>18.276</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.0399</td>
<td>13.151</td>
<td>.0449</td>
<td>6.610</td>
<td>.0463</td>
<td>4.413</td>
</tr>
<tr>
<td>.0337</td>
<td>2.760</td>
<td>.0663</td>
<td>1.428</td>
<td>.0977</td>
<td>0.985</td>
</tr>
<tr>
<td>.0885</td>
<td>10.239</td>
<td>.1990</td>
<td>5.826</td>
<td>.3077</td>
<td>4.494</td>
</tr>
<tr>
<td>.0000</td>
<td>0.003</td>
<td>.0001</td>
<td>0.002</td>
<td>.0001</td>
<td>0.001</td>
</tr>
<tr>
<td>.0674</td>
<td>5.719</td>
<td>.1326</td>
<td>3.074</td>
<td>.1954</td>
<td>2.209</td>
</tr>
<tr>
<td>1.0000</td>
<td>563.253</td>
<td>1.0000</td>
<td>527.379</td>
<td>1.0000</td>
<td>564.254</td>
</tr>
<tr>
<td>.1136</td>
<td>30.427</td>
<td>.1528</td>
<td>21.749</td>
<td>.1358</td>
<td>12.864</td>
</tr>
<tr>
<td>.0253</td>
<td>1.026</td>
<td>.0249</td>
<td>1.026</td>
<td>.0244</td>
<td>1.025</td>
</tr>
<tr>
<td>.0188</td>
<td>18.222</td>
<td>.0212</td>
<td>18.265</td>
<td>.0218</td>
<td>18.278</td>
</tr>
</tbody>
</table>

Figure 25. Queueing Network Model Results for Strategy C
<table>
<thead>
<tr>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1177</td>
<td>10.560</td>
<td>0.2235</td>
<td>5.872</td>
<td>0.3118</td>
<td>4.262</td>
</tr>
<tr>
<td>0.0336</td>
<td>2.759</td>
<td>0.0638</td>
<td>1.419</td>
<td>0.0891</td>
<td>0.967</td>
</tr>
<tr>
<td>0.0000</td>
<td>0.003</td>
<td>0.0001</td>
<td>0.002</td>
<td>0.0001</td>
<td>0.001</td>
</tr>
<tr>
<td>0.0672</td>
<td>5.714</td>
<td>0.1277</td>
<td>3.027</td>
<td>0.1782</td>
<td>2.113</td>
</tr>
<tr>
<td>0.9979</td>
<td>167.438</td>
<td>0.9627</td>
<td>133.263</td>
<td>0.9119</td>
<td>116.841</td>
</tr>
<tr>
<td>0.1133</td>
<td>30.360</td>
<td>0.1471</td>
<td>21.348</td>
<td>0.1238</td>
<td>12.511</td>
</tr>
<tr>
<td>0.0252</td>
<td>1.026</td>
<td>0.0239</td>
<td>1.023</td>
<td>0.0223</td>
<td>1.021</td>
</tr>
<tr>
<td>0.0250</td>
<td>18.334</td>
<td>0.0238</td>
<td>18.290</td>
<td>0.0221</td>
<td>18.247</td>
</tr>
</tbody>
</table>

Figure 26. Queueing Network Model Results for Strategy D
<table>
<thead>
<tr>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.1176</td>
<td>10.555</td>
<td>.2125</td>
<td>5.747</td>
<td>.2724</td>
<td>4.017</td>
</tr>
<tr>
<td>.0336</td>
<td>2.758</td>
<td>.0607</td>
<td>1.411</td>
<td>.0778</td>
<td>0.953</td>
</tr>
<tr>
<td>.0000</td>
<td>0.003</td>
<td>.0001</td>
<td>0.002</td>
<td>.0001</td>
<td>0.001</td>
</tr>
<tr>
<td>.0672</td>
<td>5.712</td>
<td>.1214</td>
<td>2.994</td>
<td>.1556</td>
<td>2.049</td>
</tr>
<tr>
<td>.9972</td>
<td>164.720</td>
<td>.9153</td>
<td>115.205</td>
<td>.7966</td>
<td>93.721</td>
</tr>
<tr>
<td>.1132</td>
<td>30.347</td>
<td>.1398</td>
<td>21.069</td>
<td>.1081</td>
<td>12.254</td>
</tr>
<tr>
<td>.0252</td>
<td>1.026</td>
<td>.0228</td>
<td>1.021</td>
<td>.0195</td>
<td>1.017</td>
</tr>
<tr>
<td>.0474</td>
<td>11.842</td>
<td>.1212</td>
<td>17.933</td>
<td>.1296</td>
<td>14.975</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.1179</td>
<td>10.581</td>
<td>.2314</td>
<td>6.057</td>
<td>.3325</td>
<td>4.575</td>
</tr>
<tr>
<td>.0337</td>
<td>2.760</td>
<td>.0661</td>
<td>1.427</td>
<td>.0950</td>
<td>0.980</td>
</tr>
<tr>
<td>.0000</td>
<td>0.003</td>
<td>.0001</td>
<td>0.002</td>
<td>.0001</td>
<td>0.001</td>
</tr>
<tr>
<td>.0674</td>
<td>5.719</td>
<td>.1322</td>
<td>3.070</td>
<td>.1900</td>
<td>2.180</td>
</tr>
<tr>
<td>1.0000</td>
<td>354.556</td>
<td>.9967</td>
<td>273.279</td>
<td>.9725</td>
<td>213.057</td>
</tr>
<tr>
<td>.1136</td>
<td>30.417</td>
<td>.1523</td>
<td>21.712</td>
<td>.1320</td>
<td>12.760</td>
</tr>
<tr>
<td>.0253</td>
<td>1.026</td>
<td>.0248</td>
<td>1.025</td>
<td>.0238</td>
<td>1.024</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.1179</td>
<td>10.581</td>
<td>.2321</td>
<td>6.076</td>
<td>.3407</td>
<td>4.702</td>
</tr>
<tr>
<td>.0337</td>
<td>2.760</td>
<td>.0663</td>
<td>1.428</td>
<td>.0973</td>
<td>0.984</td>
</tr>
<tr>
<td>0.0000</td>
<td>0.003</td>
<td>.0001</td>
<td>0.002</td>
<td>.0001</td>
<td>0.001</td>
</tr>
<tr>
<td>.0674</td>
<td>5.719</td>
<td>.1326</td>
<td>3.074</td>
<td>.1947</td>
<td>2.205</td>
</tr>
<tr>
<td>1.0000</td>
<td>541.824</td>
<td>.9998</td>
<td>446.030</td>
<td>.9963</td>
<td>362.853</td>
</tr>
<tr>
<td>.1136</td>
<td>30.417</td>
<td>.1527</td>
<td>21.748</td>
<td>.1353</td>
<td>12.849</td>
</tr>
<tr>
<td>.0253</td>
<td>1.026</td>
<td>.0249</td>
<td>1.025</td>
<td>.0243</td>
<td>1.025</td>
</tr>
<tr>
<td>.1676</td>
<td>47.798</td>
<td>.2637</td>
<td>43.193</td>
<td>.2905</td>
<td>33.565</td>
</tr>
</tbody>
</table>

Figure 27. Queueing Network Model Results for Strategies E and F
<table>
<thead>
<tr>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.1175</td>
<td>10.548</td>
<td>.2103</td>
<td>5.727</td>
<td>.2545</td>
<td>3.933</td>
</tr>
<tr>
<td>.0336</td>
<td>2.758</td>
<td>.0601</td>
<td>1.410</td>
<td>.0727</td>
<td>0.948</td>
</tr>
<tr>
<td>.0000</td>
<td>0.003</td>
<td>.0001</td>
<td>0.002</td>
<td>.0001</td>
<td>0.001</td>
</tr>
<tr>
<td>.0671</td>
<td>5.711</td>
<td>.1202</td>
<td>2.988</td>
<td>.1454</td>
<td>2.025</td>
</tr>
<tr>
<td>.9960</td>
<td>161.248</td>
<td>.9061</td>
<td>112.809</td>
<td>.7442</td>
<td>86.998</td>
</tr>
<tr>
<td>.1131</td>
<td>30.238</td>
<td>.1384</td>
<td>21.024</td>
<td>.1010</td>
<td>12.160</td>
</tr>
<tr>
<td>.0252</td>
<td>1.025</td>
<td>.0225</td>
<td>1.021</td>
<td>.0182</td>
<td>1.016</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.1179</td>
<td>10.581</td>
<td>.2318</td>
<td>6.066</td>
<td>.3335</td>
<td>4.585</td>
</tr>
<tr>
<td>.0337</td>
<td>2.760</td>
<td>.0662</td>
<td>1.428</td>
<td>.0953</td>
<td>0.980</td>
</tr>
<tr>
<td>.0000</td>
<td>0.003</td>
<td>.0001</td>
<td>0.002</td>
<td>.0001</td>
<td>0.001</td>
</tr>
<tr>
<td>.0674</td>
<td>5.719</td>
<td>.1325</td>
<td>3.072</td>
<td>.1906</td>
<td>2.183</td>
</tr>
<tr>
<td>1.0000</td>
<td>358.209</td>
<td>.9986</td>
<td>289.951</td>
<td>.9753</td>
<td>216.819</td>
</tr>
<tr>
<td>.1136</td>
<td>30.417</td>
<td>.1526</td>
<td>21.730</td>
<td>.1324</td>
<td>12.768</td>
</tr>
<tr>
<td>.0253</td>
<td>1.026</td>
<td>.0248</td>
<td>1.025</td>
<td>.0238</td>
<td>1.024</td>
</tr>
<tr>
<td>.0753</td>
<td>19.334</td>
<td>.1480</td>
<td>20.972</td>
<td>.2129</td>
<td>22.545</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.1179</td>
<td>10.581</td>
<td>.2321</td>
<td>6.077</td>
<td>.3417</td>
<td>4.720</td>
</tr>
<tr>
<td>.0337</td>
<td>2.760</td>
<td>.0663</td>
<td>1.428</td>
<td>.0976</td>
<td>0.985</td>
</tr>
<tr>
<td>0.0000</td>
<td>.003</td>
<td>.0001</td>
<td>0.002</td>
<td>.0001</td>
<td>0.001</td>
</tr>
<tr>
<td>.0674</td>
<td>5.719</td>
<td>.1326</td>
<td>3.074</td>
<td>.1952</td>
<td>2.208</td>
</tr>
<tr>
<td>1.0000</td>
<td>556.056</td>
<td>1.0000</td>
<td>490.343</td>
<td>.9992</td>
<td>409.053</td>
</tr>
<tr>
<td>.1136</td>
<td>30.417</td>
<td>.1528</td>
<td>21.749</td>
<td>.1357</td>
<td>12.860</td>
</tr>
<tr>
<td>.0253</td>
<td>1.026</td>
<td>.0249</td>
<td>1.026</td>
<td>.0244</td>
<td>1.025</td>
</tr>
<tr>
<td>.0753</td>
<td>19.334</td>
<td>.1482</td>
<td>20.989</td>
<td>.2182</td>
<td>22.857</td>
</tr>
</tbody>
</table>

Figure 28. Queueing Network Model Results for Strategy G
<table>
<thead>
<tr>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1175</td>
<td>10.552</td>
<td>0.2148</td>
<td>5.781</td>
<td>0.2769</td>
<td>4.053</td>
</tr>
<tr>
<td>0.0336</td>
<td>2.758</td>
<td>0.0614</td>
<td>1.413</td>
<td>0.0791</td>
<td>0.955</td>
</tr>
<tr>
<td>0.0000</td>
<td>0.003</td>
<td>0.0001</td>
<td>0.002</td>
<td>0.0001</td>
<td>0.001</td>
</tr>
<tr>
<td>0.0671</td>
<td>5.712</td>
<td>0.1228</td>
<td>3.003</td>
<td>0.1582</td>
<td>2.058</td>
</tr>
<tr>
<td>0.9963</td>
<td>164.279</td>
<td>0.9256</td>
<td>120.384</td>
<td>0.8097</td>
<td>97.088</td>
</tr>
<tr>
<td>1.1546</td>
<td>43.423</td>
<td>0.2172</td>
<td>35.150</td>
<td>0.2076</td>
<td>25.509</td>
</tr>
<tr>
<td>0.0252</td>
<td>1.026</td>
<td>0.0230</td>
<td>1.022</td>
<td>0.0198</td>
<td>1.018</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1179</td>
<td>10.581</td>
<td>0.2318</td>
<td>6.068</td>
<td>0.3374</td>
<td>4.639</td>
</tr>
<tr>
<td>0.0337</td>
<td>2.760</td>
<td>0.0662</td>
<td>1.428</td>
<td>0.0964</td>
<td>0.982</td>
</tr>
<tr>
<td>0.0000</td>
<td>0.003</td>
<td>0.0001</td>
<td>0.002</td>
<td>0.0001</td>
<td>0.001</td>
</tr>
<tr>
<td>0.0674</td>
<td>5.719</td>
<td>0.1325</td>
<td>3.073</td>
<td>0.1928</td>
<td>2.194</td>
</tr>
<tr>
<td>1.0000</td>
<td>361.281</td>
<td>0.9988</td>
<td>301.412</td>
<td>0.9868</td>
<td>244.849</td>
</tr>
<tr>
<td>1.1552</td>
<td>43.608</td>
<td>0.2344</td>
<td>36.928</td>
<td>0.2530</td>
<td>27.925</td>
</tr>
<tr>
<td>0.0253</td>
<td>1.026</td>
<td>0.0248</td>
<td>1.025</td>
<td>0.0241</td>
<td>1.024</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
<th>Utilization</th>
<th>Response Time (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1179</td>
<td>10.581</td>
<td>0.2321</td>
<td>6.077</td>
<td>0.3418</td>
<td>4.724</td>
</tr>
<tr>
<td>0.0337</td>
<td>2.760</td>
<td>0.0663</td>
<td>1.428</td>
<td>0.0977</td>
<td>0.985</td>
</tr>
<tr>
<td>0.0000</td>
<td>0.003</td>
<td>0.0001</td>
<td>0.002</td>
<td>0.0001</td>
<td>0.001</td>
</tr>
<tr>
<td>0.0674</td>
<td>5.719</td>
<td>0.1326</td>
<td>3.074</td>
<td>0.1953</td>
<td>2.209</td>
</tr>
<tr>
<td>1.0000</td>
<td>559.128</td>
<td>1.0000</td>
<td>501.585</td>
<td>0.9997</td>
<td>442.489</td>
</tr>
<tr>
<td>1.1552</td>
<td>43.608</td>
<td>0.2347</td>
<td>36.983</td>
<td>0.2563</td>
<td>28.224</td>
</tr>
<tr>
<td>0.0253</td>
<td>1.026</td>
<td>0.0269</td>
<td>1.026</td>
<td>0.0244</td>
<td>1.025</td>
</tr>
</tbody>
</table>

Figure 29. Queueing Network Model Results for Strategy H
## RESPONSE TIME TABLES

(in msecs)

### Number of requests = 5

<table>
<thead>
<tr>
<th>Strategies</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td># of back-ends</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1173.7067</td>
<td>1111.5368</td>
<td>1106.0955</td>
<td>1093.1209</td>
<td>1076.7827</td>
<td>1076.7826</td>
<td>1055.9013</td>
<td>1087.1987</td>
</tr>
<tr>
<td>6</td>
<td>617.087</td>
<td>503.2648</td>
<td>490.2142</td>
<td>484.078</td>
<td>428.6259</td>
<td>428.6259</td>
<td>421.2351</td>
<td>458.5127</td>
</tr>
<tr>
<td>9</td>
<td>467.0594</td>
<td>334.3705</td>
<td>315.5391</td>
<td>313.2972</td>
<td>261.8941</td>
<td>263.8941</td>
<td>249.3356</td>
<td>284.3019</td>
</tr>
</tbody>
</table>

### Number of requests = 10

<table>
<thead>
<tr>
<th>Strategies</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td># of back-ends</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2356.7506</td>
<td>2295.8904</td>
<td>2288.7749</td>
<td>2277.5511</td>
<td>2215.9679</td>
<td>2215.9679</td>
<td>2237.8885</td>
<td>2269.5116</td>
</tr>
<tr>
<td>6</td>
<td>1194.2616</td>
<td>1080.4319</td>
<td>1068.6451</td>
<td>1061.9707</td>
<td>905.9084</td>
<td>905.9084</td>
<td>956.0146</td>
<td>1005.6126</td>
</tr>
<tr>
<td>9</td>
<td>858.1617</td>
<td>701.0841</td>
<td>682.5620</td>
<td>681.4713</td>
<td>509.5228</td>
<td>509.5228</td>
<td>517.1716</td>
<td>588.9918</td>
</tr>
</tbody>
</table>

### Number of requests = 15

<table>
<thead>
<tr>
<th>Strategies</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td># of back-ends</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3543.8085</td>
<td>3482.9716</td>
<td>3475.8566</td>
<td>3466.6326</td>
<td>3339.5784</td>
<td>3339.5784</td>
<td>3424.9681</td>
<td>3456.5894</td>
</tr>
<tr>
<td>6</td>
<td>1792.7510</td>
<td>1683.2213</td>
<td>1671.5969</td>
<td>1666.8886</td>
<td>1424.3426</td>
<td>1424.3426</td>
<td>1557.2938</td>
<td>1607.0621</td>
</tr>
<tr>
<td>9</td>
<td>1255.1678</td>
<td>1108.0571</td>
<td>1090.8745</td>
<td>1089.6635</td>
<td>810.6118</td>
<td>810.6118</td>
<td>903.2192</td>
<td>985.4986</td>
</tr>
</tbody>
</table>

Figure 30. MDBS Response Times Under Various Directory Management Strategies
fully duplicated strategy, is only 810 msecs. The reason for the poor performance of the centralized strategy is, as we have expected, owing to the fact that the controller is becoming a bottleneck. For the particular numbers of requests and numbers of back-ends under consideration, the use of Strategy E will result in a mere 35% utilization of the controller. This figure is obtained by adding the first two entries in the utilization column of Figure 27. On the other hand, the use of the centralized strategy lead to a controller utilization as high as 90% which is obtained by adding the first two entries in the utilization column of Figure 23.

The partially centralized strategy (i.e., Strategy B) was consistently better than the centralized strategy. The response time improvement of the partially centralized strategy over the centralized strategy is most evident when the number of back-ends is very large. For example, when the number of back-ends is nine and the number of requests is five, the centralized strategy has a response time of 467.06 msecs and the partially centralized strategy has a response time of 334.37 msecs which is an improvement of about 40%. The reason that the improvement is more visible at larger number of back-ends is because this is precisely when the controller in the centralized strategy becomes highly utilized. In other words, the partially centralized
strategy is a more extensible strategy than the centralized strategy. Comparing corresponding entries in Figures 23 and 24, we see that the controller utilization under the partially centralized strategy is \( \frac{1}{2} \) the controller utilization under the centralized strategy.

Next, let us consider the results of using the rotating strategy, (i.e., Strategy C). As we recall, the rotating strategy is an improvement over the partially centralized strategy because the descriptor processing is shared by the back-ends and the controller, instead of being done entirely in the controller. The results bear out the fact that the rotating strategy is indeed superior to the partially centralized strategy, since it consistently outperforms the partially centralized strategy, in terms of response time over the entire range of values for the number of back-ends and the number of requests. The improvements are not dramatic. The improvement in average response time is in the order of ten or twenty milliseconds. The largest improvement occurs when the number of back-ends is the largest. This implies, of course, that the rotating strategy is more extensible than the partially centralized strategy and, hence, more preferable to us as designers of an extensible system.
Next, let us consider the results of using the rotating without controller strategy (Strategy D). The rationale of using this strategy was that it might provide some improvement over the rotating strategy because of the fact that it serves to alleviate the controller limitation problem to a greater degree. This is because the controller is no longer involved in descriptor processing. The results of Figure 30 bear this out to some degree. In fact, the rotating without controller strategy provides a better response time than the rotating strategy over the entire range of values for the number of requests and the number of back-ends. However, the improvement of this strategy over the rotating strategy is only marginal. Thus, it is seen that the improvement provided by this strategy over the rotating one is never more than about ten milliseconds. Furthermore, the percentage improvement in response time remains fairly constant over the entire range of values of the two parameters being varied. Thus, the percentage improvement is \((1104.09 - 1093.12)/1104.09\), when the number of back-ends is three and the number of requests is five. For the same number of requests and six back-ends, the percentage improvement is \((490.2142 - 484.0788)/490.2142\). Both these represent an improvement of about 11%. A look at the controller utilizations in Figures 25 and 26 tells us that the controller is indeed a little less utilized in the
rotating without controller strategy. For instance, the controller utilization when the number of back-ends is three and the number of requests is five for the rotating strategy is 16.17%. On the other hand, the controller utilization in the rotating without controller strategy for the same values of number of back-ends and number of requests is 15.13%.

Let us now consider the results of the fully duplicated strategy (Strategy E). As we recall, this strategy, along with the next two strategies, are the three strategies that employ parallel descriptor processing during the descriptors search phase. Our results indicate that such a strategy is likely to be better than all the strategies considered thus far. Comparing it with the best strategy considered so far (the rotating without controller strategy), we see that the fully duplicated strategy can lead to as much as 34% lower response time. This happens when the number of back-ends is nine and the number of requests is 15. For these particular number of back-ends and number of requests, the response time using the fully duplicated strategy is $810.6118 \text{ msecs}$ whereas the response time using the rotating without controller strategy is $1089.6635 \text{ msecs}$. The results also indicate that the improvement of the fully duplicated strategy over the rotating without controller strategy becomes more evident at larger number of back-ends and larger number of requests. In other words, the fully
duplicated strategy is more extensible than the other strategies we have considered so far.

The utilization of the disk system plays an important part in the disparity in the response time between these two strategies. The utilization of the disks in the rotating without controller strategy (the fifth row of each table in Figure 26) is a little higher than in the fully duplicated strategy (See the fifth row of each table in Figure 27). However, a small increase in the disk utilization can increase the overall response time by a large amount owing to the fact that the disk service time is large. This is one of the causes of the better response time in the fully duplicated strategy. Another reason for the better response time is the fact that the descriptor search phase now takes a shorter length of time. For instance, when the number of back-ends is nine and the number of requests is five, the descriptor search time in the rotating without controller strategy is 18.247 msecs (see last row of corresponding table in Figure 26) while it is only 14.975 msecs (see last row of corresponding table in Figure 27) in the fully duplicated strategy. This was the reason why we had expected the fully duplicated strategy to perform better than the rotating without controller strategy. The fact that the disk system response time is also improved is unexpected.
The utilization of the controller is also seen to be a little better in the case of the fully duplicated strategy. Thus, when the number of back-ends is nine and the number of requests is five, the total controller utilization is 40.09% if the rotating without controller strategy is used, while it is only 35.02% if the fully duplicated strategy is used. This also leads to some improvement in total response time.

The results for the descriptors division by attribute strategy (Strategy F) are exactly the same as for the fully duplicated strategy (Strategy E).

Next, let us consider the results for the descriptors division within attribute strategy (Strategy G). We recall that this strategy also employs parallel descriptor processing. For small number of requests, this strategy performs marginally better than the fully duplicated strategy. For large number of requests, however, the fully duplicated strategy outperforms the descriptors division within attribute strategy. Furthermore, the fully duplicated strategy outperforms the descriptors division within attribute strategy the most, when the number of back-ends is the largest. Thus, when the number of back-ends is nine and the number of requests is fifteen, the response time under the fully duplicated strategy is 810.62 ms, and the response time under the descriptors division
within attribute strategy is 903.2192 msecs. This is an improvement of 10.3%.

Finally, let us consider the results of the fully replicated strategy (Strategy H). The results indicate that it is inferior to the three strategies which employ parallel descriptor processing and superior to the other four strategies.

Consider, once again, the results of Figure 30 for the fully duplicated strategy (Strategy E). When the number of requests is five and the number of back-ends is three, the response time is 1076.78 msecs. When the number of back-ends is increased to six and the number of requests is kept unchanged, the response time improves to 428.63 msecs. That is, when the number of back-ends increase to twice of its original number, the response time of MDBS improves to better than one half of its original time. This is a surprising result. Our data placement strategy only ensures us that the doubling of the number of back-ends may halve the number of tracks to be retrieved at each back-end. Thus, we have expected that, in the best case, a doubling of the number of back-ends would lead to a halving of the response time. Hence, the results of Figure 30 are better than expected. An examination of the results in Figure 27 reveals to us the reason for the better than expected
improvement in response time. Once again, it was the response time of the disk system that played an important role in this unexpected result. When the number of back-ends is increased as indicated, it turns out that the disk utilization decreases from 99.72% to 91.53%. As a result, the disk system response time decreases from 164.72 msecs per request to 115.21 msecs per request. Thus, not only is the number of tracks to be retrieved decreased, so is the time to retrieve each track. This explains the greater-than-expected decrease in response time. Similar reasons account for the fact that when the number of back-ends is tripled to nine while keeping the number of requests constant at five, the response time is shortened to 263.8941 msecs which is 24.5% of the original time of 1076.7827 msecs (we expected that the response time can be no better than 33% of what it was when the number of back-ends was three). In other words, by choosing such a strategy for directory management in MDBS we expect that an increase in the number of back-ends in MDBS by a factor of n will cause an improvement in response time which will be better than a factor of n.

In conclusion, the results indicate that the fully duplicated strategy (Strategy E) and the descriptors division by attribute strategy (Strategy F) are the best strategies over a wide range of values for the number of
requests and the number of back-ends. The choice is now between these two strategies. The advantage of the latter strategy is that it would occupy less storage space, since descriptors are not duplicated. However, in the following section, we will study the difference between these two strategies in terms of storage requirements for typical number of directory attributes and number of descriptors per attribute. Furthermore, in another section, we will examine whether the duplication of descriptors, as in the fully duplicated strategy, is necessary for efficient handling of update requests.

4.2.4 Storage Requirements of Directory Management Strategies

In this section, we shall compare the eight strategies on the basis of their storage requirements. As we have already indicated, two types of directory based tables are necessary in MDBS. The first type of table is the descriptor-to-descriptor-id table (DDIT) and the second type of table is the augmented cluster definition table (CDT). We shall discuss for each strategy the amount of storage needed for both types of tables, in turn.
Primarily, the eight different directory management strategies differ in the amount of storage needed for storing descriptor-to-descriptor-id tables. Except for the centralized strategy, there is no difference among the different strategies in the amount of storage needed for storing the augmented cluster definition tables. Since the size of the augmented CDTs is much larger than the size of the descriptor-to-descriptor-id tables, it is expected that there will be no significant difference among the different strategies in terms of storage requirements. Let us examine them more carefully.

A. Size Estimation of the Descriptor-to-Descriptor-Id Tables (DDITs)

We assume that a database has 1 directory attributes and that there are D descriptors per attribute. Let k be the number of descriptors that can be stored in a page of b bytes. All strategies, except strategies F and G, require the following amount of storage for storing the descriptor tables.

\[ S = u(D/K)b \]

where \( u(a) \) stands for the nearest integer equal to or greater than a. In Strategies A and B, DDITs are stored entirely at the controller. Thus, the size of the tables is
S bytes. In Strategy C, these tables are duplicated in the controller and at all the back-ends. Thus, a total of \((n+1)S\) bytes are required by these tables. In Strategies D, E and H, the tables are duplicated in all the back-ends. Hence in these strategies, the total storage required by the tables is \(nS\) bytes. In Strategy F, the total space occupied by the descriptor tables is:

\[
u(D/K)bnu(i/n)\text{ bytes.}
\]

Finally, for Strategy G, the storage needed is

\[
u(D/nk)bin\text{ bytes.}
\]

B. Size Estimation of the Augmented Cluster Definition Table (CDT)

We recall that logically the augmented CDT consists of cluster definitions and corresponding secondary memory addresses. We assume that the augmented CDT is physically implemented as follows. Consider a database with \(i\) directory attributes and \(D\) descriptors per attribute. Then the augmented CDT consists of \(iD\) entries. Each entry is formed with a descriptor id followed by a list of cluster definitions and their corresponding secondary addresses. Now, the number of bytes needed to represent a cluster is

\[
u(\lg D/8)i.
\]

Here, \(\lg\) stands for the logarithm of base two. In the
ensuing discussion, we will assume that \( t \) bytes are needed to store a track address and that \( p \) bytes are needed to indicate a back end number. Finally we assume that the average cluster size is \( c \) tracks, where \( c \geq 1 \). Thus, the size, \( X \), of each CDT entry for the centralized strategy is given by
\[
(\lg D/8) + D^* (i-1)(u(\lg D/8)i+c(t+p))
\]
Thus the total size of the augmented CDT in the centralized strategy is given by \( iDX \).

In the case of every other strategy the size of an augmented CDT entry is
\[
(\lg D/8) + (cD^* (i-1)/n)(u(\lg D/8)i+t), \quad \text{if } c \leq n;
\]
\[
(\lg D/8) + D^* (i-1)(u(\lg D/8)i+ct/n), \quad \text{otherwise}.
\]
If we denote the size of an augmented CDT entry for all strategies except the centralized one as \( Y \), the total size of CDT is \( nDY \) for all strategies except the centralized one.

The following table shows the results of our study for the combined size of both tables. In the sequel, we shall refer to the combined size of the DDITs and the augmented CDT as the directory size.
<table>
<thead>
<tr>
<th>Strategy</th>
<th>Directory Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>A: $S + ID_X$</td>
</tr>
<tr>
<td></td>
<td>B: $S + niDY$</td>
</tr>
<tr>
<td></td>
<td>C: $(n+1)S + niDY$</td>
</tr>
<tr>
<td></td>
<td>D: $nS + niDY$</td>
</tr>
<tr>
<td></td>
<td>E: $nS + niDY$</td>
</tr>
<tr>
<td></td>
<td>F: $u(D/k)bin + niDYI$</td>
</tr>
<tr>
<td></td>
<td>G: $u(D/nk)bin + niDY$</td>
</tr>
<tr>
<td></td>
<td>H: $nS + niDY$</td>
</tr>
</tbody>
</table>

C. Interpretation of the Results on Sizes

Let us not proceed to perform some actual calculations on the directory sizes. We let $i$, the number of directory attributes, be 5, $b$, the page size, be 512 bytes $k$ the number of descriptors per page, be 85, $t$, the number of bytes in a track address, be 4; and $p$, the number of bytes to represent a back-end number, be 1. The number of back-ends is taken from the set $[2, 5, 8]$, the cluster size $c$ is taken from the set $[1, 2, 3, 4]$, and the number of descriptors $D$ is taken from the set $[10, 15, 20]$. 
### Figure 31. Directory Size (in kbytes) for Various Directory Management Strategies

**Number of Descriptors per Attribute = 10**

Cluster Size = 1

<table>
<thead>
<tr>
<th>Strategies</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backends</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5002.6</td>
<td>4502.7</td>
<td>4507.8</td>
<td>4505.2</td>
<td>4505.1</td>
<td>4503.2</td>
<td>4502.2</td>
<td>4502.2</td>
</tr>
<tr>
<td>5</td>
<td>5002.6</td>
<td>4502.8</td>
<td>4515.6</td>
<td>4513.1</td>
<td>4513.1</td>
<td>4502.8</td>
<td>4513.1</td>
<td>4513.1</td>
</tr>
<tr>
<td>8</td>
<td>5002.6</td>
<td>4503.0</td>
<td>4523.4</td>
<td>4520.9</td>
<td>4520.9</td>
<td>4504.5</td>
<td>4520.9</td>
<td>4520.9</td>
</tr>
</tbody>
</table>

**Number of Descriptors per Attribute = 10**

Cluster Size = 2

<table>
<thead>
<tr>
<th>Strategies</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backends</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>7502.6</td>
<td>9002.7</td>
<td></td>
<td>9007.8</td>
<td>9005.2</td>
<td>9003.2</td>
<td>9005.2</td>
<td>9005.2</td>
</tr>
<tr>
<td>5</td>
<td>7502.6</td>
<td>9002.8</td>
<td>9015.6</td>
<td>9013.1</td>
<td>9013.1</td>
<td>9002.8</td>
<td>9013.1</td>
<td>9013.1</td>
</tr>
<tr>
<td>8</td>
<td>7502.6</td>
<td>9003.0</td>
<td>9023.4</td>
<td>9020.9</td>
<td>9020.9</td>
<td>9004.5</td>
<td>9020.9</td>
<td>9020.9</td>
</tr>
</tbody>
</table>

**Number of Descriptors per Attribute = 10**

Cluster Size = 3

<table>
<thead>
<tr>
<th>Strategies</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backends</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10002.6</td>
<td>11002.7</td>
<td>11007.8</td>
<td>11005.2</td>
<td>11003.2</td>
<td>11005.2</td>
<td>11005.2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10002.6</td>
<td>13502.8</td>
<td>13515.6</td>
<td>13513.1</td>
<td>13513.1</td>
<td>13502.8</td>
<td>13513.1</td>
<td>13513.1</td>
</tr>
<tr>
<td>8</td>
<td>10002.6</td>
<td>13503.0</td>
<td>13523.4</td>
<td>13520.9</td>
<td>13520.9</td>
<td>13504.5</td>
<td>13520.9</td>
<td>13520.9</td>
</tr>
</tbody>
</table>

**Number of Descriptors per Attribute = 10**

Cluster Size = 4

<table>
<thead>
<tr>
<th>Strategies</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backends</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>12502.6</td>
<td>13002.7</td>
<td>13007.8</td>
<td>13005.2</td>
<td>13003.2</td>
<td>13005.2</td>
<td>13005.2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>12502.6</td>
<td>18002.8</td>
<td>18015.6</td>
<td>18013.0</td>
<td>18013.0</td>
<td>18002.8</td>
<td>18013.0</td>
<td>18013.0</td>
</tr>
<tr>
<td>8</td>
<td>12502.6</td>
<td>18003.0</td>
<td>18023.4</td>
<td>18020.9</td>
<td>18020.9</td>
<td>18004.5</td>
<td>18020.9</td>
<td>18020.9</td>
</tr>
</tbody>
</table>
### Number of descriptors per attribute = 15

#### Cluster size = 1

<table>
<thead>
<tr>
<th>Strategies</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>37971.4</td>
<td>34174.6</td>
<td>34179.7</td>
<td>34177.1</td>
<td>34177.1</td>
<td>34175.1</td>
<td>34177.1</td>
<td>34177.1</td>
</tr>
<tr>
<td>5</td>
<td>37971.4</td>
<td>34174.8</td>
<td>34187.6</td>
<td>34185.0</td>
<td>34185.0</td>
<td>34174.8</td>
<td>34185.0</td>
<td>34185.0</td>
</tr>
<tr>
<td>8</td>
<td>37971.4</td>
<td>34175.0</td>
<td>34195.5</td>
<td>34193.0</td>
<td>34193.0</td>
<td>34176.6</td>
<td>34193.0</td>
<td>34193.0</td>
</tr>
</tbody>
</table>

### Number of descriptors per attribute = 15

#### Cluster size = 2

<table>
<thead>
<tr>
<th>Strategies</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>56955.8</td>
<td>68346.5</td>
<td>68351.6</td>
<td>68349.0</td>
<td>68349.0</td>
<td>68347.0</td>
<td>68349.0</td>
<td>68349.0</td>
</tr>
<tr>
<td>5</td>
<td>56955.8</td>
<td>68346.7</td>
<td>68359.5</td>
<td>68456.9</td>
<td>68356.9</td>
<td>68346.7</td>
<td>68356.9</td>
<td>68356.9</td>
</tr>
<tr>
<td>8</td>
<td>56955.8</td>
<td>68346.9</td>
<td>68367.4</td>
<td>68364.8</td>
<td>68364.8</td>
<td>68348.4</td>
<td>68364.8</td>
<td>68364.8</td>
</tr>
</tbody>
</table>

### Number of descriptors per attribute = 15

#### Cluster size = 3

<table>
<thead>
<tr>
<th>Strategies</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>75940.1</td>
<td>83534.0</td>
<td>83539.1</td>
<td>83536.5</td>
<td>83536.5</td>
<td>83534.5</td>
<td>83536.5</td>
<td>83536.5</td>
</tr>
<tr>
<td>5</td>
<td>75940.1</td>
<td>102518.6</td>
<td>102531.4</td>
<td>102528.8</td>
<td>102528.8</td>
<td>102518.6</td>
<td>102528.8</td>
<td>102528.8</td>
</tr>
<tr>
<td>8</td>
<td>75940.1</td>
<td>102518.8</td>
<td>102539.3</td>
<td>102536.7</td>
<td>102536.7</td>
<td>102520.3</td>
<td>102536.7</td>
<td>102536.7</td>
</tr>
</tbody>
</table>

### Number of descriptors per attribute = 15

#### Cluster size = 4

<table>
<thead>
<tr>
<th>Strategies</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>94924.5</td>
<td>98721.5</td>
<td>98726.6</td>
<td>98724.0</td>
<td>98724.0</td>
<td>98722.0</td>
<td>98724.0</td>
<td>98724.0</td>
</tr>
<tr>
<td>5</td>
<td>94924.5</td>
<td>136690.4</td>
<td>136703.2</td>
<td>136700.7</td>
<td>136700.7</td>
<td>136690.4</td>
<td>136700.7</td>
<td>136700.7</td>
</tr>
<tr>
<td>8</td>
<td>94924.5</td>
<td>136690.7</td>
<td>136711.1</td>
<td>136708.6</td>
<td>136708.6</td>
<td>136692.2</td>
<td>136708.6</td>
<td>136708.6</td>
</tr>
</tbody>
</table>

Figure 31. (Contd.)
<table>
<thead>
<tr>
<th>Strategies</th>
<th>Back-ends</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>160002.7</td>
</tr>
<tr>
<td>5</td>
<td>160002.7</td>
</tr>
<tr>
<td>8</td>
<td>160002.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Strategies</th>
<th>Back-ends</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>240002.7</td>
</tr>
<tr>
<td>5</td>
<td>240002.7</td>
</tr>
<tr>
<td>8</td>
<td>240002.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Strategies</th>
<th>Back-ends</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>320002.7</td>
</tr>
<tr>
<td>5</td>
<td>320002.7</td>
</tr>
<tr>
<td>8</td>
<td>320002.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Strategies</th>
<th>Back-ends</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>400002.7</td>
</tr>
<tr>
<td>5</td>
<td>400002.7</td>
</tr>
<tr>
<td>8</td>
<td>400002.7</td>
</tr>
</tbody>
</table>

Figure 31. (Contd.)
The results are shown in Figure 31. The results in this figure show the directory sizes in K bytes for various number of descriptors per attribute, size of a cluster and number of back-ends.

Strategy A has the smallest storage requirements when the size of a cluster is large. However, it has the largest storage requirements when the size of a cluster is small. Because of its very large worst case storage requirements, Strategy A is not preferable for implementation in MDBS. Let us now consider the storage requirements for the remaining seven strategies. It is clear that there is no significant difference in the storage requirements for these seven strategies. In fact, the largest difference in storage requirements among these seven strategies is .05%.

Since there is no 'superior' strategy in terms of storage requirements, the strategy chosen for implementation in MDBS should be the one which is found to be superior in terms of performance. Our results have shown that Strategies E and F were the superior ones in terms of performance. In the following section, it will be seen that the duplication of descriptors, as in Strategy E, is necessary for efficient handling of some kinds of update requests. Thus, we choose to adopt Strategy E for directory management in MDBS.
4.3 The Entire Process of Request Execution

In the previous section, we described the process of directory management in MDBS. In this final section, we shall discuss the entire sequence of actions performed by MDBS in processing the four different types of requests. We shall discuss each type of request, in turn.

4.3.1 Executing a Retrieve Request

We recall that the syntax of a retrieve request in MDBS is as follows.

RETRIEVE Query Target-list [By Attribute][WITH Pointer]

The sequence of actions taken by MDBS in order to execute a retrieve has already been described in some detail in an earlier section. We shall repeat some of the discussion here, for completeness.

The controller will first parse the request and determine that it is a retrieve request. Next, the controller will broadcast the request to all the back-ends. The back-ends will perform descriptor processing and address generation under Strategy E as described in the previous section. Upon completion, each back-end has a list of secondary memory addresses of the tracks which contain the relevant records. These tracks are accessed by the
back-end. The query in the request is used to select the records from these tracks. First, the records satisfying the query are selected. If a BY-clause is specified in the retrieve request, the selected records are grouped by the values of the attribute in the BY-clause. If no BY-clause is specified in the retrieve request, all the selected records are treated as a single group. Next, for each group of selected records, the values of all attributes in the target-list are extracted from the records of the group. If no aggregate operator is specified on an attribute in the target-list, the extracted values of the group are returned to the controller. If an aggregate operator is specified on an attribute in the target-list, some computation is performed on all the attribute values in the records of the group and a single aggregate value is returned to the controller. This completes the actions performed by a back-end on each group of selected records. If a WITH-clause is specified in the retrieve request, the secondary memory addresses of all selected records must also be sent to the controller by each back-end.

The controller will wait for responses from all the back-ends. Upon receiving all the responses (i.e., attribute values, aggregate values or addresses) from all back-ends, the controller will forward these responses to the user that issued the retrieve request. This completes
the execution of the retrieve request.

4.3.2 Executing a Delete Request

As we recall, the syntax of a delete request is

```
DELETE Query
```

The execution of this request in MDBS is similar to the execution of a retrieve request. The controller will first parse the request and determine that it is a delete request. Next the controller will broadcast the request to all back-ends. The back-ends will perform descriptor processing and address generation under Strategy E. Upon completion, each back-end has a list of secondary memory addresses of tracks which contain relevant records. Records of these tracks are retrieved from the secondary memory by respective back-ends. The query in the delete request is used to select the records which are to be deleted. These selected records are then marked for deletion. The track space occupied by the marked records is not immediately recovered. Such recovery of space will be done during database reorganization time. After the records are marked, the marked records are written back to the same tracks by each back-end. If all the records in a track are marked for deletion, the address of this track is removed from all entries in which it appears in the augmented cluster.
definition table (CDT). Finally, each back-end will send an
acknowledgement to the controller to indicate that it has
finished executing the delete request. Upon receiving the
acknowledgements from all the back-ends, the controller will
inform the user or user program that the delete request has
successful y been completed.

4 3.3 Executing an Update Request

The syntax of an update request in MDBS is as follows

UPDATE Query Modifier

We recall that the modifier in an update request specifies
the new value to be taken by the attribute being modified
and that it may be one of the types described below.

Type-0 : <attribute = constant>
Type-I : <attribute = f(attribute)>
Type-II : <attribute = f(attribute-1)>
Type-III : <attribute = f(attribute-1) of Query>
Type-IV : <attribute = f(attribute-1) of Pointer>

In the simplest case, a modifier indicates the new value to
be taken by the attribute being modified (i.e., type-0). In
the more involved cases, the modifiers specify the new value
to be taken by the attribute being modified as a function f
of the 'old' value of that attribute (i.e., type-I) or values of some other attribute of the record to be updated (e.g., types-II, III or IV). The other attribute is called the base attribute (i.e., attribute-1 in the specification).

We will first describe the execution of an update request containing modifiers of types 0, I and II. We will then describe the execution of an update request containing modifiers of type-III or IV.

An update request containing a modifier of types 0, I or II is broadcast by the controller to all the back-ends. The back-ends will perform descriptor processing and address generation under Strategy E. Afterwards, each back-end has a list of secondary memory addresses of the tracks containing the relevant records. These tracks are accessed by respective back-ends and the records satisfying the query are selected from these tracks. These are the records to be updated.

Each of these records is updated using the modifier in the update request. If the modifier is of type-0, the new value to be taken by the attribute being modified in a record to be updated is provided in the modifier. If the modifier is of type-I, the new value to be taken by the attribute (being modified) in a record (to be updated) is computed as a function (specified in the modifier) of the
value of the same attribute. Finally, if the modifier is of type-II, the new value to be taken by the attribute (being modified) in a record (to be updated) is computed as a function $f$ (specified in the modifier) of the value of the base attribute in that record.

Due to its change in attribute values, an updated record may remain in the same cluster to which it (more precisely, its pre-updated version) belonged or it may now belong to a different cluster. In the latter case, a record is said to change cluster. Recall that a cluster is a group of records such that every record in the cluster is derived from the same set of descriptors. Thus, an updated record will belong to a different cluster only if the set of descriptors from which it is derived is different from the set of descriptors from which the pre-updated version was derived. If the attribute being modified in an updated record is not a directory attribute, the updated record continues to be derived from the same set of descriptors, since only directory attributes affect the descriptors. Hence, the updated record does not change cluster. If the attribute being modified is a directory attribute, an updated record may change cluster. Consider, for example, a database with the descriptors as depicted in Figure 14. Consider also the following record

$((\text{Location}, \text{NAPA}), (\text{Number}, 32123), (\text{Balance}, 50))$. 
This record is clearly derived from the descriptor set \([D2, D3, D5]\) (see Figure 14 again). After updating the value of the Balance which is a directory attribute in this record, the updated record is now shown below.

\[
(\langle \text{Location,NAPA}\rangle, \langle \text{Number,32123}\rangle, \langle \text{Balance,100}\rangle)
\]

By the way this update uses a type-0 modifier. The updated record is also derived from the same descriptor set \([D2, D3, D5]\). Hence, the record does not change cluster. On the other hand, if the value of the Balance is changed to 600, the newly updated record is not derived from the descriptor set \([D2, D3, D5]\). Instead, it is derived from the set \([D2, D4, D5]\). Hence, the record changes cluster.

In order to check whether or not a newly updated record changes cluster, it is necessary for a back-end to search the descriptor-to-descriptor-id table (DDIT). To facilitate such search, we have decided that each back-end should replicate the descriptors for all the directory attributes in its secondary memory. This is an additional reason that we decided on Strategy E. For example, if Strategy F were employed, a back-end would need to access descriptors placed at other back-ends in order to determine clusters of updated records. Consequently, bus traffic will be excessive which would create the so-called control message problem.
Finally, each back-end will send an acknowledgement to the controller to indicate that it has finished processing the update request. When it has received acknowledgements from all back-ends, the controller will output a message to the user to signal successful completion of the update request. This completes the processing of an update request containing modifiers of type-0, I or II.

Now let us describe the execution of an update request containing a type-III or IV modifier. In this case, another record must first be retrieved by MDBS on the basis of a user-provided query or pointer. After the record is retrieved, the controller will extract the base attribute value $v$ from the retrieved record. It will then compute the function $f$ (specified in the type-III or IV modifier) on the value $v$ and thus obtain a new value $v'$. The controller will then form a type-0 modifier of the form

$$<a = v'>$$

where $a$ is the attribute being modified, i.e., the same attribute appeared to the left of the equality sign in the type-III or IV modifier. The original type-III or IV modifier in the update request is now replaced with this newly created type-0 modifier. In other words, MDBS converts an update request containing a type-III or IV modifier to an update request containing a type-0 modifier. This update request containing a type-0 modifier may now be
executed in the same manner described previously.

4.3.4 Executing an Insert Request

The syntax of an insert request in MDBS is

\[ \text{INSERT Record} \]

The controller will first parse the request and determine that it is an insert request. Next, the controller will broadcast the request to all the back-ends. The back-ends will perform descriptor processing under Strategy E. The descriptor search phase under Strategy E for record insertion is more specialized than the descriptor search phase under the same strategy for other requests. For example, for the retrieve request, the descriptor search phase involves parallel descriptor processing of multiple predicates of a given query conjunction. For the insert request, the descriptor search phase merely involves parallel descriptor processing of multiple attribute-value pairs (i.e., keywords) of the record to be inserted. The specialized descriptor processing for record insertion tends to simplify the processing effort. More specifically, if the record for insertion contains \( x \) directory attributes, each back-end will determine the corresponding descriptors for \( x/n \) attribute-value pairs. At the end of the descriptor search phase, the single cluster to which the record to be
inserted is known to the back-end(s) whose secondary memory (memories) has (have) been accommodating the cluster. The reason that more than one back-end may be involved in accommodating the cluster in consideration is that the cluster being sufficiently large has been evenly distributed by the data placement strategy over several back-ends' secondary memories at the database-creation time. Consequently, MDBS must decide which back-end's secondary memory is to be used for accommodating the new record.

The address generation phase for record insertion requires an additional step under Strategy E. Instead of generating the secondary memory address immediately upon the completion of descriptor processing, each back-end sends at most one cluster id of the corresponding descriptors to the controller. By consulting a cluster-id-to-next-back-end table (CINBT), the controller can select the secondary memory of a specific back-end for record insertion. The CINBT which is depicted in Figure 32 is created for the controller at database-creation time by the data placement strategy. Upon receiving a message from the controller, the specific back-end will then continue into the address generation phase by producing a secondary address for record insertion.
This table is stored at the controller to be used during record insert and database re-organization.

<table>
<thead>
<tr>
<th>Cluster Id</th>
<th>Next Back-end for Insertion</th>
<th>Number of Tracks*</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

* Information in this and additional columns will be used for database re-organization.

Figure 32. The Cluster-Id-To-Next-Back-end Table (CINBT)
Thus, in this chapter, we have completely described the execution of the four types of requests that may be issued to MDBS.
From the previous chapter, we learn that directory management in MDBS consists of three major phases. In the first phase, MDBS determines the exact clusters which will satisfy the user request. In the second phase, MDBS accesses security information about the user to select for the user the authorized clusters among the clusters which have been determined in the first phase. In the third phase, MDBS determines the secondary memory addresses of the authorized clusters selected in the second phase. The first and third phases of directory management, known as, respectively, the descriptor search and address generation phases, have been described in detail in Chapter IV. In
this chapter, we describe the second phase of directory management. The processing performed by MDBS during the second phase of directory management will be referred to as the access control phase of directory management.

A. The Authorization Step

In describing the descriptor search and address generation phases of directory management, we note that the database creator may specify a number of descriptors at the database-creation time. These descriptors were used by MDBS to form the clusters of the database. (See Chapter III for the definition and use of descriptors.) Similarly for access control, the database creator may also specify additional information along with each descriptor at the database-creation time. The process of request execution with access control begins long before the request is issued by the user to MDBS. There is the initial step of authorization by the database creator. The authorization step requires the database creator to specify, for each potential user of the database, the specific data and intended operations which are authorized for the user. It is therefore possible that two users of the same database may be authorized by the database creator with different data and operations of the database. The role of MDBS is to
receive, maintain and enforce the authorizations. All the information that may be specified by a database creator at the database-creation time is explained in Section 5.1. How the information specified by the database creator is used by MDBS to determine the clusters which are authorized for a given user among all the clusters requested by the user is described in Section 5.2. Only after such determination is completed can a request from a user be carried through in MDBS.

B. Three Types of Access Control — by Granules, by Statistics and by Values

Upon receiving a user request, MDBS begins the process of request execution with access control. In controlling access to the database, MDBS provides an access control mechanism with considerable capabilities. First, it protects clusters. In other words, only authorized clusters will be permitted for the user. Further, only authorized operations are performed on the clusters so authorized. Second, it protects attribute values in the records of a cluster. Thus, access control is now brought from the cluster and record level to the field (i.e., attribute-value pair) level. This provides a finer granularity of security. Third, it protects the statistics of the database by
controlling the execution of requests which utilize the aggregate functions such as average, maximum, minimum and others. Thus, statistical security becomes an integral part of the access control mechanism. Finally, it protects data on the basis of the relationship of the attribute value of the data and the user of the data. For example, a user may be given access to the records of those and only those employees who are managed by the user. Obviously, this user is a manager. Note that such access control cannot be achieved by protecting the values of some attributes in the records using the fine granularity of security discussed earlier. This is because if we were to give all employees who are managers an access to those records then we would make the records available to other managers. What is really needed is for this and only this manager to access records of his or her employees. In other words, the access control is dependent upon specific attribute values. The access control mechanism of MDBS provides value-dependent security. In Section 5.4, we will describe two extensions to the basic access control mechanism for statistical security and value-dependent security. In Section 5.5, we describe the process of execution of transactions (consisting of multiple requests) with access control. Finally, in Section 5.6, we consider how a database creator may modify the authorization that was specified at the
C. A New Mode of Operation - Precision Control by Multiple Back-ends

From our subsequent discussion of the access control mechanism of MDBS, we will note the following two characteristics. First, the access control mechanism operates in a distributed fashion across multiple back-ends rather than in the controller. This serves to alleviate the controller limitation problem. Second, we will note that, in MDBS, all access control checks are made prior to the retrieval of records from the secondary store. Consequently, access imprecision due to redundant record retrieval never occurs. Thus, the need to discard retrieved records due to security violations does not arise, since only authorized records are retrieved from the secondary memory. Let us assume for example, that a user wishes to retrieve 21 records but that 5 of these 21 records are not authorized for the user. Then, a conventional access control mechanism will first retrieve the 21 records. It will examine these 21 records to eliminate the 5 unauthorized ones before providing the user with the 16 authorized ones. However, in MDBS, the new access control mechanism will ensure that only the 16 authorized records
will be retrieved from the secondary memory. Consequently, the access precision, defined as the ratio of the number of authorized records vs. the number of retrieved records, is always absolute, i.e. equal to 1, for MDBS.

5.1 Access Control as Exercised by the Database Creator

In this section, we will describe the kinds of access control information which may be specified by the database creator at the database-creation time. We will also describe how the information is stored in MDBS. In order to control access, a database creator needs to specify who can perform what operations on which data in his or her database. What we, as system designers, would like to do is to provide the database creator with an effective, yet efficient, way to convey these three pieces of information to MDBS. It is straightforward for a database creator to specify two of the three pieces of information, i.e., to identify the users who are allowed to access the database and to specify for these users the kinds of accesses allowed (or disallowed) to the database. For specifying 'who', the database creator utilizes the user id provided by the MDBS. For specifying 'what', the database creator selects the intended access operations from the set of MDBS operations such as retrieve, insert and so on. The third piece of
information, to specify 'which' portions of the database is to be authorized for the users may be more involved. Let us describe the method used in our system for specifying portions of the database to be controlled for access.

We begin by arguing that a cluster is an ideal unit for access control. In Chapter IV, we showed that a cluster is the basic unit of access in MDBS. This is because the clusters are formed in such a way that whenever a user requests some records of a cluster, there is a high likelihood that all the records in the cluster are needed together. Since all the records in a cluster are likely being accessed together, it makes sense to accord the same control to all the records belonging to the cluster. Hence, we decide that a cluster should be a unit of access control in MDBS.

However, a database creator is unaware of the formation of clusters by MDBS. The database creator is only aware of the descriptors which in turn induce the clusters. Therefore, the database creator is provided with additional means to specify access control information with respect to these descriptors. Since these descriptors are used to form the clusters, the access control information specified by the database creator with respect to the descriptors may be transformed by MDBS into access control information about
the clusters of the database. Thus, we now have a method for controlling access to clusters. If a cluster consists of a single record, then MDBS may control access at the level of individual records. If a cluster consists of many records, then MDBS may control access to record aggregates. Finally, MDBS may control certain attribute values of the records in a cluster. It will be seen that our access control mechanism is capable of three levels of control. In the following paragraphs, we will describe the means with which the database creator may specify various levels of access control.

First of all, a creator of a database must decide the users who will be allowed to access the database, i.e. the users who are allowed to issue requests (from either a terminal or an application program) to the database using the data manipulation language (DML) provided in Chapter III. Note that the database creator is also one of the users of the database. After deciding the number of users, $p$, allowed to access the database, the database creator will inform MDBS of this number. MDBS will then assign $p$ user ids for these $p$ users. The database creator will reserve one of these ids for his or her own use and pass on the remaining ids to the other $(p-1)$ users of the database. A user trying to use MDBS from a terminal must first identify the user by supplying the assigned id. Similarly, a user
submitting a program to be executed in MDBS must include the user id as a part of the program. In this way, any request received by MDBS, either from a user at a terminal or as a part of a user program, is associated with the user. We shall refer to this id as the user id of the request.

Having received the id of the users of the database, the database creator now specifies a number of descriptors as described in Section 4.1.1. In addition, the creator specifies, together with each descriptor, p sets of field level access controls, one for each user of the database. A field-level access control is either null or 'all' or a pair of the form

\[(\text{attribute combination}, \text{disallowed access operation})\].

An attribute combination is one or more attributes of the database. A disallowed access operation is one of the set \{No-Retrieve, No-Delete, No-Insert, No-update\}. In the sequel, we shall often refer to the attribute combination part of a field-level access control and to the disallowed access (operation) part of a field-level access control. Whenever the disallowed access part of a field-level access control is No-Delete or No-Insert, the attribute combination part is left unspecified. This is because we always insert and delete entire records in MDBS. An example of a descriptor and a set of field-level access controls for a user is as follows:
The meaning of this example is as follows. The user is not permitted to update the job information from salary records whose salaries are ranged exclusively between zero and 100. Nor is the user permitted to retrieve the name and salary information from the same salary records. More formally, MDBS prevents the user from any update of the value of the job attribute in those records whose keywords (i.e., salary attribute and salary value pairs) are derivable from the descriptor \(0<\text{Salary}<100\). Similarly, MDBS prevents the user to read values of the Name and Salary attributes from records whose keywords are derivable from the descriptor \(0<\text{salary}<100\). We also stated that a field-level access control could be either null or all which indicates, respectively, that either no access or all accesses are disallowed. This completes our description of the kinds of information that must be specified by the database creator at the database-creation time.

By way of an example, let us illustrate the kinds of information specified by a database creator. Consider the database of employee records shown in Figure 33 and let us assume that only two users, identified as user 1 and user 2, are allowed to access this database. We assume that user 1 is the database creator and that user 2 is the only other
One database with six records and four attributes per record

\[ R_1: (\text{Employee,1}, \text{Department,1}, \text{Salary,1000}, \text{Manager,1}) \]
\[ R_2: (\text{Employee,2}, \text{Department,1}, \text{Salary,8000}, \text{Manager,1}) \]
\[ R_3: (\text{Employee,3}, \text{Department,2}, \text{Salary,15000}, \text{Manager,3}) \]
\[ R_4: (\text{Employee,4}, \text{Department,2}, \text{Salary,2000}, \text{Manager,3}) \]
\[ R_5: (\text{Employee,5}, \text{Department,3}, \text{Salary,7000}, \text{Manager,5}) \]
\[ R_6: (\text{Employee,6}, \text{Department,3}, \text{Salary,16000}, \text{Manager,5}) \]

Figure 33. A Sample Database for Illustrating Field-Level Access Control
user_allowed_to_access_the_database.

After having received the user_ids (i.e., user_1 and user_2) from MDBS, the database-creator specifies a number of descriptors. Furthermore, corresponding to each descriptor, the database creator specifies two sets of field level access controls for the two users of the database, respectively. This information specified by the database creator is stored in an augmented descriptor-to-descriptor-id table (DDIT). (See a description of DDIT in Section 4.1.1.) The augmented DDIT consists of p additional columns, where p is the number of users anticipated to access the database. Thus, each entry of DDIT now consists of (p+2) fields. The first field contains a descriptor id and the second field contains a descriptor. The third field contains the set of field-level access controls for the first user corresponding to the descriptor that is in the second field. Similarly, the (p+2)-th field contains the set of field-level access controls for the p-th user corresponding to the same descriptor. An augmented DDIT for the sample database of Figure 33 is shown in Figure 34. We see that four descriptors (identified as D1, D2, D3 and D4) have been specified for the database. We also see that two sets of field-level access controls have been specified for each of these descriptors. In this figure, the set consisting of
<table>
<thead>
<tr>
<th>id</th>
<th>Descriptor</th>
<th>Set of Field-Level Access Controls For User 1</th>
<th>Set of Field-Level Access Controls For User 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>(1000 ≤ Salary ≤ 10000)</td>
<td>-</td>
<td>([Employee, Salary], No-Retrieve)</td>
</tr>
<tr>
<td>D2</td>
<td>(10000 &lt; Salary &lt; ∞)</td>
<td>-</td>
<td>All</td>
</tr>
<tr>
<td>D3</td>
<td>(1 ≤ Department ≤ 2)</td>
<td>-</td>
<td>([Manager], No-Update), ([Employee, Salary], No-Retrieve)</td>
</tr>
<tr>
<td>D4</td>
<td>(Department = 3)</td>
<td>-</td>
<td>([Manager], No-Retrieve)</td>
</tr>
</tbody>
</table>

'-' indicates that no access is disallowed
'All' indicates that all accesses are disallowed

Figure 34. The Augmented DDIT for the Sample Database of Figure 33.
only the null field-level access control is denoted by '-' which indicates that no access is disallowed. It is therefore clear that user 1, the database creator, is not disallowed any access. That is, he is allowed to access any portion of the database. User 2, on the other hand, may access the database only in a controlled manner. In referring to Figure 34 again, for example, he is not allowed to learn the names of the managers from records in Department 3. This is controlled by the descriptor D4. The set of field-level access controls for user 2 corresponding to descriptor D2 is shown to be 'All' in Figure 34. This is used to indicate that all accesses are disallowed for user 2 to salary records whose salaries are greater than 10,000.

We have now completely described the kinds of information specified by a database creator, and we have also indicated how this database-creator-specified information is stored in an augmented DDIT.

5.2 Determination and Organization of the Exact Access Control from the Database-Creator-Specified Information

Once the descriptors and corresponding sets of field-level access controls have been specified and stored, the clusters of the database may be formed by using the
algorithm for cluster formation described in Section 4.1.1. The clusters formed for the sample database of Figure 33 are reflected in the cluster definition table (CDT) as depicted in Figure 35.

5.2.1 Controlling Access to Authorized Clusters

In order to control access, we also need, corresponding to each cluster, $p$ sets of field-level access controls, one for each user of the database. The set of field-level access controls for a user corresponding to a cluster indicates the disallowed accesses for the user to that cluster. We now describe how the set of field-level access controls for a user corresponding to a cluster is determined.

Let us assume that we wish to find the set of field-level access controls for user 2 on cluster 2. We first learn, from Figure 35, that cluster 2 is defined by the descriptors D1 and D4. Then, the set of field-level access controls for user 2 on cluster 2 is obtained as the union of the sets of field-level access controls for user 2 corresponding to descriptors D1 and D4. The set of field-level access controls for user 2 on descriptors D1 and D4 are obtained from the augmented DDIT (see Figure 34).
<table>
<thead>
<tr>
<th>Cluster id</th>
<th>Set of Descriptors</th>
<th>Records in the Cluster Defined by the Descriptor Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{D1, D3}</td>
<td>R1, R2, R4</td>
</tr>
<tr>
<td>2</td>
<td>{D1, D4}</td>
<td>R5</td>
</tr>
<tr>
<td>3</td>
<td>{D2, D3}</td>
<td>R3</td>
</tr>
<tr>
<td>4</td>
<td>{D2, D4}</td>
<td>R6</td>
</tr>
</tbody>
</table>

Figure 35. The Cluster Definition Table (CDT)
Thus, their union can be taken to determine the set of field-level access controls for user 2 on cluster 2. In general, the set of field-level access controls for user $x$ on cluster $y$ is obtained as the union of the sets of field-level access controls for user $x$ corresponding to each of the defining descriptors of cluster $y$. By repeating the above procedure for every cluster in the database and every user authorized for the database, the sets of field-level access controls for each user over every cluster may be determined.

5.2.2 Organizing and Storing Cluster Control Tables

Having determined the set of field-level access controls for each user over every cluster, we are now concerned with methods for organizing and storing the access control information. There are two possible techniques for organizing and storing the access control information. The first technique utilizes a separate cluster definition table for each user of the database. The table for a user will only contain the access control information of those clusters to which the user is allowed some access. This table contains one more column than the CDT described in Chapter IV. This additional column contains the set of field-level access controls for this user corresponding to
the various clusters. For the examples developed in Figures 34 and 35, the augmented CDT for user 2 is shown in Figure 36. Note that there are only two clusters in the augmented CDT for user 2, even though there are four clusters in the database. To summarize, such a technique requires MDBS to maintain one CDT for the entire database and an augmented CDT for each user.

A second technique for organizing and storing the set of field-level access controls for each user over every cluster is to collect all the access control information for all the users in one centralized cluster definition table. In this case, the sets of field-level access controls over every cluster is stored in MDBS by augmenting the CDT with \( p \) columns, one for each user of the database. In our example, the CDT of Figure 35 is augmented by two columns as shown in Figure 37. This centralized CDT will then be used by MDBS during request execution.

Either technique may be used in MDBS for organizing or storing the set of field-level access controls for each user over every cluster. The first technique will generally be superior in terms of the speed of request execution. This is because the number of clusters in the augmented CDT for a user will generally be less than the number of clusters in the centralized CDT for all the users. Consequently the
<table>
<thead>
<tr>
<th>id</th>
<th>Set of Descriptors</th>
<th>Records in the Cluster Defined by the Descriptor Set</th>
<th>Set of Field-Level Access Controls</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{D1, D3}</td>
<td>R1, R2, R4</td>
<td>({Manager}, No-Update),\n{Employee, Salary}, No-Retrieve)</td>
</tr>
<tr>
<td>2</td>
<td>{D1, D4}</td>
<td>R5</td>
<td>({Manager}, No-Retrieve),\n{Employee, Salary}, No-Retrieve)</td>
</tr>
</tbody>
</table>

Figure 36. The Augmented Cluster Definition Table for User 2
### Figure 17. The Augmented CDT for the Sample Database of Figure 33

<table>
<thead>
<tr>
<th>Cluster id</th>
<th>Set of Descriptors</th>
<th>Records in the Cluster Defined by the Descriptor Set</th>
<th>Set of Field-Level Access Controls For User 1</th>
<th>Set of Field-Level Access Controls for User 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{D1,D3}</td>
<td>R1,R2,R4</td>
<td>-</td>
<td>{([Manager],No-Update), ([Employee,Salary],No-Retrieve)}</td>
</tr>
<tr>
<td>2</td>
<td>{D1,D4}</td>
<td>R5</td>
<td>-</td>
<td>{([Manager],No-Retrieve), ([Employee,Salary],No-Retrieve)}</td>
</tr>
<tr>
<td>3</td>
<td>{D2,D3}</td>
<td>R3</td>
<td>-</td>
<td>All</td>
</tr>
<tr>
<td>4</td>
<td>{D2,D4}</td>
<td>R6</td>
<td>-</td>
<td>All</td>
</tr>
</tbody>
</table>

'-' indicates that no access is disallowed

'All' indicates that all accesses are disallowed
first technique entails a search through a smaller table. This is what contributes to the speed superiority of the first technique.

The first technique is also superior in terms of ease of maintenance. For instance, if one of the users of the database is removed from the system, we may simply delete the corresponding augmented CDT. In the second technique, we need to remove all the field-level access controls corresponding to that user from the centralized CDT. Irrespective to the physical implementation of the centralized CDT, the removal of all the field-level access controls corresponding to a user from the centralized CDT can be no simpler than the deletion of a table. Thus, we conclude that the first technique should be chosen in MDBS for organizing and storing the sets of field-level access controls for each user over every cluster. That is, physically, a separate augmented CDT is maintained for each user of the database, in addition to the one CDT for the entire database which is also maintained by MDBS. Logically, however, we may still assume that the sets of field-level access controls for each user over every cluster are consolidated and maintained as a single centralized CDT. Such an assumption simplifies some of the ensuing discussion.
In the next section, we will describe the process of request execution with access control which makes use of the augmented or centralized CDT formed at the database-creation time.

5.3 Request Execution With Fine Granularity of Access Control

Up to this point, all the events we have described take place at the database-creation time long before any request is being executed. Let us now describe the process of request execution with access control for insert and non-insert requests in MDBS. We recall again (see Chapter 11) that the process of request execution consists of the three phases of directory management followed by actual processing of records retrieved from the secondary memory. Let us refer to the last stage of request execution in MDBS, i.e., the one follows the three phases of directory management, as record processing. In Section 4.2.1, we referred to the first and third phases of directory management as the descriptor search phase and the address generation phase, respectively. We also referred to the processing performed during the descriptor search phase as descriptor processing and the processing performed during the address generation phase as address generation. In
summary, the process of request execution in MDBS without access control consists of descriptor processing, address generation and record processing. When access control is required, the process of request execution has added complexity. It still consists of descriptor processing and record processing as described in Section 4.3. However, the address generation phase is expanded.

In Chapter IV, we stated that address generation is executed at each back-end and consists of two steps. In the first step, the corresponding descriptors produced by the descriptor processing are used in order to find the corresponding cluster (for an insert request) or the corresponding set of clusters (for a non-insert request). We also stated that in the second step the secondary memory addresses of the corresponding cluster or corresponding set of clusters are generated. When access control is enforced, address generation is again executed at each back-end, but it consists of one additional step. Let us discuss this step with other steps together. In the first step, the corresponding descriptors produced by the descriptor processing are used in order to find the corresponding cluster or set of clusters. This step has not been changed. In the second step, the authorized clusters are selected on the basis of the information provided in the augmented CDT. This is the new step. Finally in the third step, the
secondary memory addresses of the corresponding authorized clusters are generated. In this final step, the amount of processing may be reduced because the number of authorized clusters is usually smaller than the number of relevant clusters. However, the processing logic does not change. To summarize our discussion, the process of request execution with access control is different from the process of request execution without access control because the former consists of one additional step during address generation. We now elaborate the processing performed in this additional step of address generation as follows.

5.3.1 Executing Insert Requests

In the case of an insert request, MDBS performs descriptor processing and the first step of address generation in order to determine the cluster $k$ and the back-end $b$ into which the record in the request is to be inserted. (See again Section 4.3.4.) After such determination, the controller sends the record to back-end $b$ for insertion. The second (i.e., new) step of address generation is now activated. The back-end $b$ will search its augmented CDT in order to determine if the user that issued the insert request is authorized with such an insert to cluster $k$. This is accomplished by looking up the entry in
the row for cluster \( k \) and in the column for the user id of the user that issued the insert request. If this entry contains a field-level access control with an unspecified attribute combination part and with 'No Insert' in the disallowed access part, the request is rejected. Otherwise, the insert request is permitted. Consequently, the back-end proceeds with the next step of address generation and final step of record processing (both of which have been described in Section 4.3).

5.3.2 Executing Non-Insert Requests

Non-insert requests consist of retrieve, delete and update. Before we describe the process of request execution for these requests, we will discuss some implications of their execution in the context of access control.

We recall that the database creator specifies descriptors and several sets of corresponding field-level access controls. These field-level access controls indicate the disallowed access operations. Thus, for instance, the database creator may specify the following descriptor and corresponding set of field-level access controls for user \( l \).

\[
(0 < \text{Salary} \leq 100) \quad ([\text{Salary}], \text{No-Retrieve})
\]

This indicates that user \( l \) is not permitted to retrieve the
salary field from records containing keywords derived from the descriptor \((0<\text{Salary} \leq 100)\). However, the database creator has not indicated whether user 1 is allowed to update, delete and insert such records. Therefore, the MDBS access control mechanism must make a choice in this case for each of the three unspecified access operations - update, delete and insert. A naive access control mechanism may permit user 1 to update, delete or insert records containing keywords derived from the descriptor \((0<\text{Salary} \leq 100)\), since such operations have not been specifically disallowed in the field-level access controls by the database creator. However, such naive access control would permit user 1 to compromise the access control of the database. We illustrate the problem of compromised access control with the following example.

A. A Case of Compromised Access Control Due to Alternative Operation

Let us consider a salary database in which a typical and single-attribute-valued record \(R_1\) is

\[ R_1: (\langle \text{Salary}, 50 \rangle) \]

Let the database creator specify the following descriptor and corresponding field-level access control for user 1.

\[(0<\text{Salary} \leq 100) ([\text{Salary}], \text{No-Retrieve})\]
Let cluster 1 consist of all the records whose keywords are derived from the descriptor \((0<\text{Salary} \leq 100)\). Then, it follows that the set of field-level access controls for user 1 corresponding to cluster 1 is

\[
([\text{Salary}], \text{No-Retrieve})
\]

That is, user 1 is not permitted to retrieve the salary field from records belonging to cluster 1. However there is no indication as to whether user 1 is permitted to update or delete (or insert) the (a) salary field from (to) records belonging to cluster 1. Let us assume that a naive access control mechanism permits user 1 to update the salary field of records belonging to cluster 1. For instance, the following update request by user 1 is allowed to take place.

\[
\text{UPDATE} \ (\text{Salary} = 50) <\text{Salary} = \text{Salary} + 0>
\]

If the salary database does not contain a record for an employee earning 50, the mechanism will return a negative acknowledgement to the user. On the other hand, if the database does contain a record for an employee earning 50, the mechanism will return a positive acknowledgement to the user. Thus, user 1 can easily make the inference whether there is an employee record in the database with earning being 50. In other words, user 1 may infer at the information which he would have obtained if he had issued the retrieve request

\[
\text{Retrieve} \ (\text{Salary} = 50) \ (\text{Salary})
\]
since user 1 is not allowed to issue the above retrieve request as dictated by the field-level access controls specified by the database creator, the user obtains the salary information of the employee by way of an update request. Any access control mechanism which allows information to be revealed to a user by way of one type of requests while the same information is expressly prohibited from being revealed to the user under the other types of requests is said to suffer from the problem of compromised access control as characterized in the above case study. Obviously, the MDBS access control mechanism must not have the problem of compromised access control.

In the example, the problem of compromised access control arises because user 1 is allowed to update the value of an attribute from records in a specific cluster, even though he is not allowed to retrieve the value of this attribute from records in the cluster. Clearly, the problem may be overcome by a more sophisticated access control mechanism which ensures the following rule. Whenever a user is not allowed to retrieve the value of an attribute from records in a cluster, the user must also not be allowed to update the value of that attribute from records in the cluster. We term such a rule, the rule of access control implication. We therefore say that the No-Update disallowed access operation is implied by the No-Retrieve disallowed
access operation. Graphically, we denote the implication as follows.

\[
\text{No-Retrieve} \rightarrow \text{No-Update}
\]

For each user, the MDBS access control mechanism will enforce not only the set of field-level access controls over every cluster induced by the database creator's specification, but also all the other disallowed access operations implied by disallowed access operations in the field-level access controls.

B. Two More Cases of Compromised Access Control Due to Alternative Operation

In addition to the case of compromised retrieval due to update, there are other implications. Disallowed update operations may be compromised by delete operations, since No-Delete is implied by No-Update. The converse is also true. Consider, once again, the salary database of the previous example. Let us assume that the database creator specifies the following descriptor and corresponding field-level access control for user 1.

\[
(0 < \text{Salary} \leq 100) \ (\left[ \text{Salary} \right], \text{No-Update})
\]

Furthermore, let cluster 1 consist of all the records whose keywords are derived from the descriptor \((0 < \text{Salary} \leq 100)\). Then, it follows that the set of field-level access control
for user 1 corresponding to cluster 1 is

\( ([\text{Salary}], \text{No-Update}) \).

That is, user 1 is not allowed to update the salary field of the records belonging to cluster 1. However, there is no indication of whether user 1 is allowed to retrieve the salary field from records belonging to cluster 1. Nor is there any indication of whether user 1 is allowed to insert new salary into or delete old salary from such records. To illustrate how a naive access control mechanism may compromise the database by allowing user 1 to insert and delete records of cluster 1, we propose the following scenario.

Knowing that update operation is denied from the user, user 1 tries the following requests in sequence.

- DELETE \((\text{Salary} = 50)\)
- INSERT \(\langle\text{Salary}, 52\rangle\)

Thus, user 1 is able to achieve the effect of updating the salary by 2. The access control is compromised because the user is disallowed to update any salary of records in cluster 1, and yet the user is able to achieve the effect of update by means of delete and insert operations. Clearly, the problem of compromised access control in the example may be overcome by either disallowing the insert operation or the delete operation on records in cluster 1. In general, a sophisticated access control mechanism should recognize the
following implications

No-Update -> No-Delete

No-Update -> No-Insert

Conversely, we learn that if a user is not allowed to delete records from a cluster, the user must not be allowed to update attribute values of records in the cluster. That is, the implication

No-Delete -> No-Update

holds. To illustrate the necessity of upholding the implication, we present the following counterexample. Assume that the field-level access control corresponding to cluster 1 for user 1 is

([], No-Delete),

where the '[]' indicates that the attribute combination part is unspecified. Since the update operation on records in cluster 1 has not been specifically disallowed in the field-level access control, a naive access control mechanism may allow user 1 to update all the fields of records in cluster 1. Thus, user 1 may achieve the effect of deleting a record by updating the value of every field in the record to null. Hence, the problem of compromised access control exists.
We conclude that in order to overcome the problem of compromised access control, a sophisticated access control mechanism must enforce the following four implications:

(1) No-Retrieve $\rightarrow$ No-Update
(2) No-Retrieve $\rightarrow$ No-Delete
(3) No-Update $\rightarrow$ No-Delete
(4) No-Delete $\rightarrow$ No-Update

Now, we are ready to describe the process of request execution with access control for non-insert requests. We have already described, in Section 4.1.3, how descriptor processing and the first step of address generation are used to determine the set of clusters corresponding to the query in the user request. Now, we need to do the following for each cluster $k$ in this set in order to exercise access control. By looking up an entry in the augmented CDT corresponding to the row for cluster $k$ and the column for the user id of the user that issued the non-insert request, each back-end will make the following checks.

If the request is a delete request, each back-end will check to see if the entry of the augmented CDT contains a field-level access control with any attribute combination part and with 'No-Delete', 'No-Retrieve' or 'No-Update' in the disallowed access part. By checking for 'No-Retrieve' in the disallowed access part, the implication
No-Retrieve $\rightarrow$ No-Delete

is being upheld. Similarly, by checking for 'No-Update' in the disallowed access part, the implication

No-Update $\rightarrow$ No-Delete

is being upheld. Since both the implications related to the delete request are thus upheld by each back-end, the problem of compromised access control does not occur in MDBS during the execution of the delete request.

If the request is an update request, each back-end will check to see if the entry of the augmented CDT contains a field-level access control with the attribute being modified as one of the attributes in the attribute combination part and the disallowed access part as 'No-Update' or 'No-Retrieve'. For instance, if the attribute being modified is salary, it will check to see if the entry contains a field-level access control of the form ([Salary,...],No-Update) or ([Salary,...],No-Retrieve), where the attribute combination part may have other attributes besides salary. By checking for 'No-Retrieve' in the disallowed access part, the implication

No-Retrieve $\rightarrow$ No-Update

is being upheld. Each back-end will also check to see if the entry contains a field-level access with an unspecified attribute combination part and with 'No-Delete' in the disallowed access part. This allows the back-end to uphold
the implication

\[ \text{No-Delete} \rightarrow \text{No-Update} \]

Since both the implications related to the update request are thus upheld by each back-end, the problem of compromised access control cannot occur in MDBS during the execution of the update request.

If the request is a retrieve request, each back-end will check to see if the entry of the augmented CDT contains a field-level access with the attribute combination part containing one or more attributes from the target-list (of the retrieve request) and the disallowed access part being 'No-Retrieve'. For example, consider the retrieve request

\[ \text{RETRIEVE (File = EMPLOYEE) (Name,Salary)} \]

In this case, each back-end will check to see if the entry contains a field-level access control of the form

\[ ([\text{Name},.] , \text{No-Retrieve}) \text{ or } ([\text{Salary},...], \text{No-Retrieve}) \]

Here, we have only described the processing of retrieve requests which do not contain aggregate operators in their target-lists. The processing of requests with aggregate operators is described later in Section 5.4.

If the check is positive in any of the cases mentioned above, the back-ends delete cluster \( k \) from the set of corresponding clusters. In other words, cluster \( k \) is a cluster relevant to the user request; nevertheless, it is
not authorized for the user. The above procedure is repeated for every cluster in the set of corresponding clusters. The remaining set of clusters is referred to as the permitted or authorized set of clusters for the non-insert request. Once the set of permitted clusters for a non-insert request have been determined, the back-ends proceed with the third step of address generation and with record processing as described in Section 4.3.

5.3.3 The 'Conservative' and Precision Access Control Mechanism

First, we would like to point out that the MDBS access control mechanism may be 'conservative' in that it may reject a user request by 'overly' protecting the requested data. For instance, consider the following retrieve request issued by user 1.

RETRIEVE (Dept=TOY) (Salary)

For simplicity, assume that all the relevant records of the toy department belong to cluster 1 and that the set of field-level access control for user 1 corresponding to cluster 1 has been determined as

([Name, Salary].No-Retrieve).

In this case, Salary is the only attribute in the target-list of the retrieve request. Also, Salary is one of
the attributes in the attribute combination part of the field-level access control for user 1 corresponding to cluster 1. Thus, according to the algorithm presented in the previous section, the access to the salary field is denied to user 1 and the request is rejected. However, it might be the intention of the database creator that user 1 is only to be disallowed from reading the name and salary fields jointly from records in cluster 1 in order to prevent the user from learning individual employee salaries. That is, the database creator might intend to allow user 1 to access either the name or salary fields singly but not jointly from records in cluster 1, since separate lists of names and salaries might not correlate the individual employee salaries. In view of this example, MDBS access control mechanism might be considered to be conservative in not allowing user 1 to access the salaries from such records.

However, there is a very good reason for the conservatism of the MDBS access control mechanism. Consider that, in the above example, user 1 were allowed to read the name and salary fields individually from records in cluster 1. Then, user 1 may issue the following two requests one after the other.

RETRIEVE (Dept=TOY & SS =50) (Name), and
RETRIEVE (Dept=TOY & SS =50) (Salary),
As a result, the user may obtain the name and salary of the employee in the toy department where social security number is 50. But this may be precisely the information that the database creator may intend to disallow user 1 from obtaining. In other words, if the MDBS access control mechanism is not 'conservative', it is possible for users to compromise the access control of the database. It is to overcome the problem of compromised access control that MDBS has a 'conservative' access control mechanism.

Second, we note that MDBS access control mechanism may eliminate one or more clusters from consideration in the second step of address generation. Since a cluster is stored in one or more tracks, the elimination of a cluster from consideration will result in savings in terms of reduced number of accesses to the secondary store.

We also note that in MDBS all access control checks are made prior to the retrieval of records from the secondary store. Consequently, access imprecision due to redundant record retrieval never occurs. Thus, the need to discard retrieved records due to access control violations does not arise since they are retrieved from the secondary memory only after they are cleared for access.
5.3.4 An Example of the Process of Request Execution With Access Control

In order to illustrate the entire process of request execution with access control, we consider the database of Figure 33 and the corresponding augmented CDT of Figure 37. Let us assume that the user identified as user 2 issues the request

\textsc{RETRIEVE (Salary < 10000) (Manager)}.

This request requires the retrieval of the names of managers of all those employees who earn less than $10,000. After descriptor processing and the first step of address generation, MDBS determines that the corresponding set of clusters for the query (Salary < 10000) is the set consisting of cluster 1 and cluster 2. In the second step of address generation, the access control takes place. In this step, the permitted set of clusters must be determined. The two clusters in the set of corresponding clusters are checked, in turn. Cluster 1 is checked for authorization by looking up the entry corresponding to cluster 1 and user 2 in Figure 37. There is the following entry

\[((\text{[Manager], No-Update}), (\text{[Employee Salary], No-Retrieve}))\]

which is checked to see if it contains a field-level access control of the form [Manager, ...], No-Retrieve). Since no
such field-level access control is present in the entry for cluster 1, cluster 1 is a permitted cluster. Next, cluster 2 is checked for authorization by looking up the entry corresponding to cluster 2 and user 2. There is the following entry

```
((Manager, No-Retrieve), (Employee, Salary), No-Retrieve)).
```

which is checked to see if it contains a field-level access control of the form ((Manager, ...), No-Retrieve). Since such a field-level access is indeed present in the entry for cluster 2, cluster 2 is not a permitted cluster. Thus, cluster 1 is the only permitted cluster. Then, the third step of address generation and record processing are performed for the retrieve request. Looking at Figure 36 again, we see that records R1, R2 and R4 are retrieved, since these are the only records in cluster 1. Finally, the value of the Manager attribute is retrieved from these records and presented to user 2. This completes the processing of the retrieve request for user 2.
5.4 New Capabilities of the Access Control Mechanism

In this section, we describe two extensions to the basic access control mechanism. These optional extensions are discussed in turn.

5.4.1 Statistical Access Control

It may be necessary to disallow a user from obtaining an aggregate value of some attribute over a number of records. Aggregate attribute values result from the use of aggregate functions such as average, summation, maximum and minimum. Our access control mechanism may be extended to provide such a capability as described below.

We recall that a field-level access control is a pair of the form (attribute combination, disallowed access operation), where an attribute combination is a set of attributes. We now extend our definition of attribute combination to be a set of elements. Each element is either an attribute e.g., Salary or an aggregate operator to be performed on an attribute, e.g., AVG(Salary). An aggregate operator is one of AVG, SUM COUNT, MAX, MIN. An example of an attribute combination with two elements is [Department, AVG(Salary)].
Let us now demonstrate how attribute combinations with aggregate operators may be used to control access over requests for aggregate values. Consider, for example, an employee database in which the database creator wishes to disallow user 1 from obtaining the average salary of all those employees who are working in the toy department. However, the database creator allows the same user to obtain the average salary of all those employees who are working in the sales department. Finally, the database creator disallows user 1 from obtaining the individual salaries of employees in either of these two departments. Then, the database creator may specify the following two descriptors and corresponding set of field-level access controls.

(Department=TOY) \([\text{AVG}(\text{Salary}), \text{Salary}], \text{No-Retrieve}\)

(Department=SALES)\([\text{Salary}], \text{No-Retrieve}\)

For simplicity, let cluster 1 be defined by the single descriptor

(Department=TOY).

Also, let cluster 2 be defined by the single descriptor

(Department=SALES).

Then, it is easily determined that the set of field-level access controls for user 1 corresponding to cluster 1 is \([\text{AVG}(\text{Salary}), \text{Salary}], \text{No-Retrieve}\). Similarly, the set of field-level access controls for user 1 corresponding to cluster 2 is determined as
Now assume that user 1 issues the request

\[ \text{RETRIEVE (Department=T\textsc{oy}) (AVG(Salary)).} \]

MDBS will first perform descriptor processing and the first step of address generation to determine that the only cluster corresponding to the query in the user request is cluster 1. Since the set of field-level access controls for user 1 corresponding to cluster 1 disallows access to the salary average, the request from user 1 is rejected. On the other hand, assume that user 1 issues the request

\[ \text{RETRIEVE (Department=SA\textsc{les}) (AVG(Salary)).} \]

In this case, MDBS will first perform descriptor processing and the first step of address generation to determine that the only cluster corresponding to the query in the user request is cluster 2. Since the set of field-level access control for user 1 corresponding to cluster 2 does not disallow access to the salary average, the request is not rejected. MDBS will determine the secondary memory addresses of the records in this cluster and retrieve these records. It will then extract the values of the salary attribute from such records compute the average and output a single value for the average salary. This average salary will then be returned to user 1. Now consider that user 1 issues the request
RETRIEVE (Department=TOY) V (Department=SALES) (AVG(Salary)).

In this case, MDBS will perform descriptor processing and the first step of address generation to determine that the set of clusters corresponding to the query in the request is the set consisting of clusters 1 and 2. In the second step of address generation, MDBS will determine the set of permitted clusters. Since the set of field-level access controls for user 1 corresponding to cluster 1 does not allow access to the salary average, cluster 2 is the only permitted cluster. Now MDBS may take one of two possible courses of action. One possibility is to access the records in cluster 2, extract the values of the salary attribute from such records, compute a salary average and return this average salary to the user. However, the average salary returned to user 1 is restricted to the employees in the sales department. What the user had requested was the average salary of all employees in both the sales and toy departments. Therefore, MDBS must also return a message to user 1 indicating that the value returned is only an authorized one. The second possible course of action is simply to reject the request, since the requested average touches upon unauthorized data. Either course of action may be chosen for our implementation.
A. An Example of Compromised Statistical Access Control Due to Users' Own Aggregate Operations

Consider a salary database and let the database creator specify the following descriptor and corresponding set of field-level access control for user 1.

\[(0<\text{Salary} \leq 100) \land ([\text{AVG(Salary)}], \text{No-Retrieve})\]

That is, the database creator specifies that user 1 is not allowed to access the salary average of all records containing keywords derived from the descriptor \((0<\text{Salary} \leq 100)\).

Let us assume that cluster 1 is defined by this same descriptor. Then, it is easy to see that the set of field-level access control for user 1 corresponding to cluster 1 is

\([\text{AVG(Salary)}], \text{No-Retrieve})\]

That is, user 1 is not allowed to access the salary average of the records in cluster 1. However, there is no indication of whether the user is allowed to access the individual salary field of such records. Consider a naive access control mechanism which allows user 1 to access the salary field from records in cluster 1. Now, user 1 may access the salary field from each and every record in cluster 1. Then, the user may compute the average of all these retrieved salaries (either manually or by running a statistical program). In this way, user 1 may obtain the
average salary of all records in cluster 1, thus compromising the access control of the database. Therefore, a sophisticated access control mechanism will not permit user 1 to access the salary field from records in cluster 1.

In order to ensure that the statistical security of the database is not compromised, MDHS access control mechanism incorporates the following additional procedure in the process of request execution with access control for retrieve requests.

B. An Expanded Procedure for Effective Statistical Access Control

We recall that the attribute combination part of a field-level access control and the target-list of a retrieve request are both lists of elements where each element is either an attribute or an aggregate operator to be performed on an attribute. Whenever an element in an attribute combination part of a field-level access control is an aggregate operator on an attribute, the attribute is also added to the attribute combination part of the field-level access control. Thus for example, the field-level access control

\[ ([\text{AVG}(\text{Salary}, \text{Dept}), \text{No-Retrieve})], \]
becomes in effect

\[(AVG(Salary), Salary Dept), No-Retrieve)\].

Every field-level access control in the relevant entry of the augmented CDT is expanded in this way. Now, each back-end will check to see if the entry of the augmented CDT contains a field-level access control with the attribute combination part containing one or more elements from the target-list of the retrieve request. If the check is positive, each back-end will delete cluster \( k \) from the set of corresponding clusters. The above procedure is repeated for every cluster in the set of corresponding clusters. Thus, we form the set of permitted clusters for the retrieve request. Then, each back-end may proceed with the final step of address generation and with record processing as described in Section 4.3.

5.4.2 Value-Dependent Access Control

In this section, we will describe how MDBS may control access to data on the basis of the relationship of the attribute value of the data and the user of the data. In this case, the access control is dependent on the values of a specific attribute. Thus, we refer to it as value-dependent access control. Furthermore, we shall refer to the specific attribute whose values are used to enforce
access control as the relationship attribute. Let us illustrate the value-dependent access control with an example.

Consider that the database creator wishes to give a user access to the records of those and only those employees who are managed by the user. In this case, the relationship attribute is Manager. With respect to the sample database of Figure 33, employee 1 may only access the first two records, employee 3 may only access the next two records, and employee 5 may only access the last two records.

In this case, value-dependent access control is implemented in MDBS as follows. As before, the database creator specifies a number of descriptors and corresponding set of field-level access controls at the database-creation time. In addition, the database creator specifies a relationship attribute and a set of field-level access controls corresponding to that attribute. This single relationship attribute and single set of field-level access controls must be converted by the system into a number of descriptors and corresponding sets of field-level access controls. For simplicity, we will select from the descriptors specified by the database creator at the database-creation time the only relationship attribute and the corresponding set of field-level access controls for
consideration. In our example, we want to disallow a user all accesses to records of employees whom the user does not manage. That is, the only relationship attribute is Manager and the set of field-level access controls is the one which disallows all accesses. In the sequel, we shall designate this set of field-level access controls with 'All'.

MDBS must convert the relationship attribute and set of field-level access controls to a number of descriptors and p sets of field-level access controls for each descriptor, where p is the number of potential users of the database. Once this conversion is achieved, the algorithm described in Section 5.2 may be used to form the augmented cluster definition table. Then, this table may be used as before during request execution with access control. In the next paragraph, we explain how the relationship attribute and set of field-level access controls are converted to a number of descriptors and p sets of field-level access controls.

The conversion is illustrated with the use of the sample database of Figure 33. We assume that each employee in the database of Figure 33 is permitted to access the database and that these six employees are the only users of the database. Thus, p is set to six. The conversion algorithm is as follows. First, we extract all possible values of the relationship attribute from the records of the
database and place it in a set V. In our example, we will extract all possible values of the Manager attribute from the records of Figure 33 and form the set V as \{1, 3, 5\}. For each value v in the set V, we will form a descriptor of the form

\[(\text{relationship attribute} = v)\]

In our example, we will form the three descriptors \((\text{Manager} = 1)\), \((\text{Manager} = 3)\) and \((\text{Manager} = 5)\). Corresponding to each descriptor so formed, we need to form p sets of field-level access controls. In our example, corresponding to each of the three descriptors, we need to form six sets of field-level access controls. The set of field-level access controls corresponding to user i for a descriptor of the form \((\text{relationship attribute} = v)\) is set to 'All' (the database creator specified set of field-level access control) if i is not equal to v. Otherwise, it is set to null to indicate that no access is disallowed. The descriptors and corresponding sets of field-level access controls formed for our example are shown in Figure 38.

This completes our description of the algorithm for converting from the database-creator-specified relationship attribute and corresponding set of field-level access controls to a number of descriptors and p sets of field-level access controls where p is the number of potential users of the database.
<table>
<thead>
<tr>
<th>Descriptor</th>
<th>Set of Field-Level Access Controls for User 1</th>
<th>Set of Field-Level Access Controls for User 2</th>
<th>Set of Field-Level Access Controls for User 3</th>
<th>Set of Field-Level Access Controls for User 4</th>
<th>Set of Field-Level Access Controls for User 5</th>
<th>Set of Field-Level Access Controls for User 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Manager=1)</td>
<td>—</td>
<td>All</td>
<td>All</td>
<td>All</td>
<td>All</td>
<td>All</td>
</tr>
<tr>
<td>(Manager=3)</td>
<td>All</td>
<td>All</td>
<td>—</td>
<td>All</td>
<td>All</td>
<td>All</td>
</tr>
<tr>
<td>(Manager=5)</td>
<td>All</td>
<td>All</td>
<td>All</td>
<td>All</td>
<td>—</td>
<td>All</td>
</tr>
</tbody>
</table>

'All' indicates that all accesses are disallowed
'-' indicates that no access is disallowed

Figure 38. The Descriptor and Corresponding Sets of Field-Level Access Controls Formed for the Database of Figure 33
Once the descriptors and corresponding sets of field-level access controls are obtained, the algorithm described in Section 5.2 may be used to create the augmented cluster definition table and the algorithm of Section 5.3 is used to enforce access control during request execution.

5.5 Access Control for Transactions of Multiple Requests

Up to this point, we have described the process of request execution with access control for individual requests. However, as described in Chapter III, we also allow for transactions of multiple requests to be carried out in MDBS. In this section, we consider the process of transaction execution with access control. Two techniques are considered for executing transactions consisting of multiple requests. The first technique, called the stand-alone execution is known for its simplicity. The second technique, called attached execution is more complex. Nevertheless, it is more effective and efficient in controlling access of transactions to the database.
5.5.1 Stand-Alone Execution of Transactions

Each request in a transaction is executed in the manner described in Section 5.3. Thus, no special attention is paid to the fact that these requests are actually parts of a transaction.

5.5.2 Attached Execution of Transactions

Each request in a transaction is executed till the point where the set of authorized clusters for the request are determined. That is, descriptor processing and the first two steps of address generation are performed on each request in the transaction. Now, each request in the transaction may be attached with the authorized set of clusters for that request. We shall refer to an attached request as a secure request because access control processing has been completed for this request. By forming a secure request from every request in a transaction, we may form a secure transaction from the original transaction. The secure transaction formed from the transaction is referred to as the secure version of the transaction. Note that each request in a secure transaction has only been partially but not completely executed. After a secure transaction has been formed, each request in the secure
transaction may then be executed to its completion by executing the third step of address generation followed by record processing.

5.5.3 The Choice of Transaction Execution for Access Control

There are two reasons that we prefer to select the attached execution technique for controlling access of transactions to the database. In order to explain the first reason, we define the permitted set of clusters for a transaction as the union of the authorized set of clusters for each request in the transaction. In Chapter VI, we will see that, to enforce concurrency control, a transaction must lock its set of permitted clusters before any access to the secondary memory may be made on behalf of the transaction. Therefore, it is necessary to preprocess transactions to determine their sets of permitted clusters. This is precisely what we do in the attached execution technique described above. On the other hand, in the stand-alone execution technique, a transaction may be partially executed even before its set of permitted clusters is entirely determined. If we were to employ the stand-alone execution technique, then no lock for concurrency control could be placed on the permitted set of clusters for a transaction
since such a set is not known beforehand.

The second reason for preferring the attached execution technique has to do with the fact that transactions in MDBS are meant to be executed repeatedly. If this technique is employed, we need to enforce access control for the requests in a transaction only the first time the transaction is executed. After this first execution of a transaction, a secure version of it may be preserved for use during subsequent executions of this transaction. On the other hand, if the stand-alone execution technique is employed, it is necessary to enforce access control each time a transaction is executed, since a secure version of the transaction is not available.

It should be emphasized, nevertheless, that in the attached execution technique the preserved secure version of a transaction may be used in subsequent executions of that transaction only as long as the database creator has not made any changes to the access control information about the database. If the database creator does make modifications to the access control information, the preserved secure versions of transactions are no longer valid and must be recalculated.
5.6 The Management of Access Control Information

From time to time a database creator may change the access control information for his or her database. We shall consider the various ways in which a database creator may change the access control information and the various actions to be taken by MDBS in response to these changes. Here, we find it necessary to talk at the implementation level rather than at the logical level of the access control information. Accordingly, we keep in mind that in addition to the augmented descriptor-to-descriptor-id table (DDIT), there is the augmented CDT which is physically implemented as one cluster definition table (CDT) and separate cluster definition tables (SCDTs), one for each user of the database. Let us now proceed to discuss the various changes that a database creator may wish to make to the access control information.

5.6.1 Denying a User From Any Access to the Database

Consider that the database creator wishes to disallow one of the users from accessing the database. In that case, each back-end will simply delete the SCDT for that user. Thus, the actions taken by MDBS in response to this change made by the database creator are simple.
5.6.2 Adding a New User of the Database

Let us consider that the database creator wishes to add a new user to the system. In this case, he must first acquire a new user id from MDBS which the database creator will pass on to the new user. Next, the database creator needs to specify one set of field-level access controls for each descriptor in the database. These sets of field-level access controls are stored as a new column of the augmented DDIT. Pictorially, the augmented DDIT of Figure 34, before and after the addition of a new user (i.e., user 3) is shown in Figure 39. In the figure, F31, F32, F33 and F34 are the sets of field-level access controls specified for the new user. After a new column has been added to the augmented DDIT, each back-end must next form an SCDT for the new user. This requires the back-end to determine the sets of field-level access controls for user 3 on every cluster. The algorithm described in Section 5.2 is used to do this. Once the SCDT for user 3 is created, all requests issued by user 3 may be executed in the manner described in Section 5.3.
Table: Field-Level Access Controls for Different Users

<table>
<thead>
<tr>
<th>Descriptor Id</th>
<th>Descriptor</th>
<th>Set of Field-Level Access Controls for User 1</th>
<th>Set of Field-Level Access Controls for User 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>(1000 ≤ Salary ≤ 10000)</td>
<td>F₁₁</td>
<td>F₂₁</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D4</td>
<td>(Department = 3)</td>
<td>F₁₄</td>
<td>F₂₄</td>
</tr>
</tbody>
</table>

New Augmented DDIT Due to Changes of Above

<table>
<thead>
<tr>
<th>Descriptor Id</th>
<th>Descriptor</th>
<th>Set of Field-Level Access Controls for User 1</th>
<th>Set of Field-Level Access Controls for User 2</th>
<th>Set of Field-Level Access Controls for User 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>(1000 ≤ Salary ≤ 10000)</td>
<td>F₁₁</td>
<td>F₂₁</td>
<td>F₃₁</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D4</td>
<td>(Department = 3)</td>
<td>F₁₄</td>
<td>F₂₄</td>
<td>F₃₄</td>
</tr>
</tbody>
</table>

Fᵢⱼ: Set of Field-level Access Controls for User 𝑖 Corresponding to Descriptor 𝑗

Figure 39. The Augmented DDIT of Figure 33 Before and After the Addition of User 3 to the System
5.6.3 Changing the User's Access Operations Only

The database creator may wish to change the corresponding set of field-level access controls for a user on a descriptor. Then, MDBS performs the following operations. First, it makes the appropriate changes to the augmented DDIT. For instance, let us assume that we wish to change the set of field-level access controls for user 2 on descriptor Dl from ([Employee, Salary], No-Retrieve) as shown in Figure 34 to ([Employee], No-Retrieve). The augmented DDIT of Figure 34, after appropriate changes have been made to it, is now shown in Figure 40. After appropriate changes have been made to the augmented DDIT, each back-end will then modify the SCDT of the user whose set of field-level access controls was changed by the database creator. In general, the new sets of field-level access controls for the user correspond only to some of the clusters. Thus, only the access control information of these clusters needs to be modified in the user's SCDT. Specifically, in our example, the sets of field-level access controls, for user 2 corresponding to all those clusters whose definitions contain Dl, need only to be modified. This is done by recalculating the sets of field-level access controls for these clusters using the algorithm of Section 5.2. The original SCDT, before modification, for user 2 is the one shown in Figure 36. The modified SCDT for user 2 is shown
<table>
<thead>
<tr>
<th>Id</th>
<th>Descriptor</th>
<th>Set of Field-level Access Controls For User 1</th>
<th>Set of Field-level Access Controls For User 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(1000 ≤ Salary ≤ 10000)</td>
<td>—</td>
<td>([Employee], No-Retrieve)</td>
</tr>
<tr>
<td></td>
<td>(10000 &lt; Salary &lt; ∞)</td>
<td>—</td>
<td>All</td>
</tr>
<tr>
<td></td>
<td>(1 ≤ Department ≤ 2)</td>
<td>—</td>
<td>([Manager], No-Retrieve),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>([Employee, Salary], No-Retrieve))</td>
</tr>
<tr>
<td></td>
<td>(Department = 3)</td>
<td>—</td>
<td>([Manager], No-Retrieve)</td>
</tr>
</tbody>
</table>

'-' indicates that no access is disallowed

'All' indicates that all accesses are disallowed

Figure 40. The Augmented DDIT After Changing The Set of Field-Level Access Controls for User 2 on Descriptor D1
in Figure 4.1. Once the SCDT for user 2 is modified, all requests issued by user 2 may executed in the manner described in Section 5.3.

5.6.4 Changing the User's Permitted Data Granules

There are four other changes which a database creator may wish to make. A database creator may wish to add a new descriptor, delete an existing descriptor, coalesce two existing descriptors into a single descriptor, or split up an existing descriptor into two descriptors. Since descriptors are used to define clusters, changes in descriptors will cause changes in cluster definitions and in the sets of field-level access controls corresponding to clusters. Thus, it will be necessary to delete the existing SCDTs and to create new SCDTs. Thus, each of these changes on the part of the database creator will require some work on the part of MDBS.

A. Defining New Granules With New Descriptors

Consider, first, the addition of a new descriptor. In this case, the database creator also needs to specify p sets of field-level access controls, one set for each user
(See Figure 36 for the original SCDT for User 2)

<table>
<thead>
<tr>
<th>Cluster id</th>
<th>Set of Descriptors</th>
<th>Records in the Cluster Defined by the Descriptor Set</th>
<th>Set of Field-level Access Controls</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( { D_1, D_4 } )</td>
<td>( R_1, R_2, R_4 )</td>
<td>( { [\text{Manager}], \text{No-Update} }, { [\text{Employee}], \text{No-Retrieve} } )</td>
</tr>
<tr>
<td>2</td>
<td>( { D_1, D_4 } )</td>
<td>( R_5 )</td>
<td>( { [\text{Manager}], \text{No-Retrieve} }, { [\text{Employee}], \text{No-Retrieve} } )</td>
</tr>
</tbody>
</table>

Figure 41. The New Separate Cluster Definition Table (SCDT) for User 2
allowed to access the database. Now, a row for this new descriptor may be added to the augmented DDIT. Then, the algorithm presented in Section 5.2 is used to determine the sets of field-level access controls for every user on every cluster. In this way, several new SCDTs are created at each back-end, one for every user allowed to access the database. This completes the actions to be taken by MDBS in response to the addition of a new descriptor by the database creator.

B. Creating Larger Granules By Deleting Descriptors

Let us consider what happens when one of the existing descriptors is deleted from MDBS. First, the row corresponding to this descriptor will be deleted from the augmented DDIT. Then, the algorithm presented in Section 5.2 is used to determine the sets of field-level access controls for every user on every cluster. Thus, several new SCDTs are created at each back-end to replace the deleted SCDTs. Thus, the actions to be taken by MDBS in this case are very similar to the actions taken by MDBS when a new descriptor is added.
C. Creating Larger Granules by Coalescing Descriptors

Let us consider that the database creator wishes to coalesce two descriptors into a single descriptor. For instance, he may wish to coalesce the descriptor \((0 < \text{Salary} \leq 100)\) and the descriptor \((101 < \text{Salary} < 200)\) into the single descriptor \((0 < \text{Salary} < 200)\). In this case, the database creator also needs to specify \(p\) sets of field-level access controls for the new single descriptor, one set for each user allowed to access the database. MDBS responds by first appropriately modifying the augmented DDIT. This modification consists of the removal of the rows corresponding to the two descriptors being coalesced and the addition of a row for the new single descriptor. Then, the algorithm of Section 5.2 may be used by each back-end to create the new SCDTs of the database to replace the deleted ones.

D. Defining New Granules By Splitting Descriptors

The final case we consider is when a descriptor needs to be split up into two descriptors. Consider that the database creator wishes to split descriptor \(D\) into descriptors \(D_1\) and \(D_2\). This may be considered as the deletion of descriptor \(D\) followed by the addition of
descriptors D1 and D2. Since the actions to be taken by MDBS during descriptor addition and descriptor deletion have already been specified, the actions to be performed by MDBS during descriptor splitting is also specified.
CONCURRENCY CONTROL FOR MULTIPLE BACK-ENDS AND CONSISTENCY OF PARTITIONED DATABASES

In this chapter, we will first raise the issue whether MDBS indeed needs any form of concurrency control. Once we are convinced that we need some form of concurrency control, we will then examine the various existing concurrency control mechanisms in terms of MDBS requirements. In the examination, we discover that a new requirement for consistency known as monolithic consistency, will have to be met. We will therefore strive for a simple concurrency control mechanism for upholding the monolithic consistency in MDBS. This chapter will mostly be devoted to the design and illustration of the MDBS concurrency control mechanism.
6.1 Is there a Necessity for Concurrency Control?

In a multiple back-end database system, there is no need for concurrency control as long as each back-end is neither executing multiple requests in a concurrent fashion nor executing requests from multiple users in an interleaved fashion. Let us elaborate this observation with the following discussion. We recall that a transaction consists of multiple requests. Let transaction t1 consist of requests r1 and r2, and let transaction t2 consist of requests r3 and r4. If a back-end executes r3, after r1 has been completely executed, then we say that the back-end is executing requests (i.e., transactions) from multiple users in an interleaved manner. We note that in this case there is no concurrent execution of requests. A back-end is said to be executing two requests in a concurrent manner if both requests have been partially executed by the back-end, but neither request has been completely executed. Is it necessary for a back-end to be executing requests in a concurrent and in an interleaved fashion?

The argument against interleaved execution of requests from multiple users and against concurrent execution of multiple requests for the same or different user is that it obviates the need for having a concurrency control mechanism at each back-end, thereby making the software at each
back-end simpler. The designer of the distributed database system, known as Stonebraker's machine [Ston78], makes this argument with the following additional points. It is felt that the back-ends should not support concurrent request execution, because concurrent request execution would not be beneficial to the system performance when only one disk drive is attached to a back-end. In [Ston78], it was also argued that concurrent request execution will prove advantageous only if multiple disk drives are connected to each back-end.

However, we argue strongly that MDBS should support concurrent and interleaved execution of user requests and therefore concurrency control for the following reasons.

A. The Throughput Issue

The disadvantage of not allowing for concurrent request execution is that the throughput of MDBS will be poorer. This is because database requests in MDBS are mostly I/O-bound. Thus, a back-end will be mostly waiting for the disk to access and retrieve data. The use of clusters and placement strategy in our system ensures us that only a small amount of processing will be needed after the data is retrieved. Thus, the idle time is likely to be high unless the back-end utilizes the time for the execution of
concurrent requests.

B. The Response-Time Issue

We also believe that interleaved execution of user requests should also be supported at the back-ends of MDBS. The reasons for supporting interleaved execution of user requests is that, otherwise, the response time to a user's request will be high.

C. The Multiple Disk Drive Issue

There are two good reasons for attaching multiple disk drives to a minicomputer back-end. First, minicomputers are powerful enough to handle multiple disk drives. For example, a PDP-11/44 is designed to operate with up to 64 disk drives. If a back-end were to be attached to only a single disk drive, it would appear to be functioning like a disk controller. In fact, even a conventional disk controller is capable of managing multiple disk drives. Thus, we would be making poor use of the processing power of a minicomputer if we were to attach only a single disk drive to it. The second reason for having more than one disk drive at each back-end is that we are trying to support very large databases. We cannot support very large databases by attaching only one disk drive to a back-end.
From the above arguments, we conclude that MDBS should support both concurrent and interleaved execution of user requests. We therefore endeavor to devise a concurrency control mechanism for such support.

6.2 **What are the Necessary and Sufficient Conditions for a Consistent Partitioned Database Utilizing Multiple Back-ends?**

Concurrency control mechanisms are developed for the purpose of upholding the consistency of the database. For example, in centralized databases, concurrency control mechanisms must ensure inter- and intra-consistencies of data [Hsia81]. In distributed databases, on the other hand, concurrency control mechanisms must ensure, in addition to inter- and intra-consistencies, mutual consistency [Thom79]. We first argue that mutual consistency of multiple copies of the same data is not necessary for a partitioned database such as ours. We note that only the augmented descriptor-to-descriptor-id-table (DDIT) is duplicated in MDBS. In other words, the same augmented DDIT is stored at each back-end of MDBS, whereas the augmented cluster definition tables (CDTs) and the database are not duplicated at the back-ends. Only the distinct augmented CDT and different clusters of the database are stored at individual back-ends. Furthermore, update requests issued
by a user will never cause any modification of the augmented DDIT. The remaining data which may be modified by the update requests are stored as a partitioned database and requires no copy of the same data. We also argue with the following counterexample that inter-consistency of data is not sufficient for a partitioned database. Thus, the necessary and sufficient conditions for a consistent database in a multiple back-end system will not be found in the conventional solutions to centralized and distributed database systems. We must identify the new consistency problem for partitioned database systems.

Let us show by means of examples, the exact nature of the problem in the context of MDBS Consider that the following two updates are issued to MDBS.

UPDATE (File=EMPLOYEE) <Salary=Salary+2>
UPDATE (File=EMPLOYEE) <Salary=Salary*2>

The first request increments the salaries of all employees by two dollars. The second request doubles the salaries of all employees. Clearly, if we allow a back-end to concurrently execute these two requests, inconsistent results may arise. Thus, some employees will have their salaries doubled; others will have their salaries incremented by two dollars; others may have their salaries changed from an original value, say \( x \), to \( (2x+2) \); and still
others may have their salaries changed from an initial value $x$ to a final value $2(x+2)$. These inconsistencies go by various names such as the problem of lost updates and the non-reproducibility of reads [Gard77]. Two requests, such as those above, whose simultaneous (or concurrent) execution can lead to inconsistencies are termed incompatible requests. More formally, we say that two requests $r_1$ and $r_2$ are compatible if their simultaneous execution gives the same result as would be obtained if their execution sequence is $r_1$ followed by $r_2$ or $r_2$ followed by $r_1$. In order to ensure that such inconsistencies do not occur, a concurrency control mechanism must ensure that two incompatible requests are executed one after the other. A mechanism which ensures serial execution of incompatible requests is said to maintain inter-consistency [Hsia81].

It is clear that if each back-end is to execute requests, one after the other, instead of concurrently no inconsistency can result. When we generalize from single requests to transactions of multiple requests, then it is clear that if each back-end is to execute transactions one after the other and also execute all the requests within a transaction one after the other, no inconsistency can result. Since any concurrency control mechanism for a centralized database ensures inter-consistency, one possible approach would be to have a centralized concurrency control
mechanism in the controller of MDBS. The controller would have a scheduler that would choose the next request for execution. This request would then be broadcasted to all the back-ends for execution. We discard this approach from consideration because it aggravates the controller limitation problem which we singled out in Chapter II for multiple back-end systems.

Another alternative would be to incorporate a centralized concurrency control mechanism at each back-end. Such an approach alleviates the controller limitation problem. However, we are questioning whether such an approach is sufficient for a partitioned database with multiple back-ends such as MDBS.

Let us recall that in MDBS a request is broadcasted to all the back-ends. Also, let us recall that MDBS executes requests in an MIMD fashion. That is, a request is not executed at exactly the same instant in all the back-ends. Rather, the request is being executed asynchronously in the different back-ends. With these observations, let us consider the following two requests which are broadcast by MDBS to each back-end.

U1: UPDATE (File=EMPLOYEE) <Salary=Salary+2>

U2: UPDATE (File=EMPLOYEE) & (Dept=6) <Salary=Salary*2>

We also consider for our example that an update U0 has been
broadcasted by MDBS to the back-ends prior to either U1 or U2. Let U0 be

U0: UPDATE (File=EMPLOYEE) & (Dept=7) <Salary=Salary+2>

We see that U0 and U1 are incompatible because they may simultaneously try to update the records belonging to department 7. However, updates U0 and U2 are compatible because they update different sets of records. Now, let there be two back-ends in MDBS. When updates U1 and U2 are received at back-end 1, let us assume that update U0 has not been completed at that back-end. Hence, back-end 1 cannot execute U1, since U0 and U1 are incompatible. However, it may start executing U2, since U0 and U2 are compatible. Back-end 1 may execute U1 at some later time after U0 and U2 have been completed. Hence, the order of execution of updates U1 and U2 at back-end 1 is U2 followed by U1. On the other hand, let us assume that U0 has completed execution at back-end 2 when U1 and U2 are received. This is possible because U0 is not executed at exactly the same time instant at both the back-ends. Furthermore, U0 may incur less processing at back-end 2 due to fewer employee records stored at that back-end. Thus, back-end 2 may execute U1. It will then execute U2 at some later time. Thus, the order of execution of updates at back-end 2 is U1 followed by U2. In spite of the fact that inter-consistency is maintained at each back-end, the result at back-end 1 is
computed in terms of \((value \times 2) + 2\) and the result at back-end 2 is computed in terms of \(((value + 2) \times 2)\). These results are likely to be different. Consequently, the state of the MDBS database is inconsistent.

From this illustration, it is obvious that the guarantee of inter-consistency at each back-end is not sufficient to guarantee the consistency of a partitioned database utilizing multiple back-ends.

6.3 Monolithic Consistency and Non-Permutable Requests

From the previous example, we make three more observations. We first observe that the inconsistent state of the MDBS database would not occur if the entire database were stored at a single back-end and the requests issued to MDBS were executed by that single back-end in some serial order. We also observe that the inconsistency occurred because the two back-ends executed two requests in different orders and because different execution orders for these two requests lead to different results at these back-ends. Thus, the inconsistency may be avoided by the following techniques. Whenever two requests \(r_1, r_2\) are such that the result of executing \(r_1\) followed by \(r_2\) is different from the result of executing \(r_2\) followed by \(r_1\), we term them
non-permutable requests and ensure that these two requests are executed in the same order at all back-ends. Thus, to maintain database consistency for multiple back-ends, we say that we must maintain the same execution order among all non-permutable requests at all back-ends. Finally, we observe that inter-consistency must also be maintained at all back-ends. Otherwise, a back-end will try to simultaneously execute incompatible requests and this leads to problems of lost updates, etc. [Gard77].

In summary, we need to maintain partitioned database consistency for multiple back-end systems such as MDBS. This is termed monolithic consistency. In order to preserve monolithic consistency we need to maintain an execution order among non-permutable requests and to preserve inter-consistency at all back-ends. To maintain an order among non-permutable requests, we need to identify pairs of requests that are non-permutable. To preserve inter-consistency, we need to identify pairs of requests that are incompatible. The identification of request pairs that are non-permutable and incompatible constitutes our major contributions in the following sections. First, however, let us develop some notations and terminology.
6.4 Notations and Terminology

In order to simplify the ensuing discussion, we use the following notation. Insert requests are of the form

\[ \text{INSERT } R \]

where \( R \) is some record to be inserted. Delete requests are of the form

\[ \text{DELETE } Q \]

where \( Q \) is the query used to select the records to be deleted. Retrieve requests are of the form

\[ \text{RETRIEVE } Q \text{ } \langle A \rangle \]

where \( Q \) is the query used to select the records to be retrieved and \( \langle A \rangle \) is the target-list of attributes whose attribute values will be fetched from the records retrieved. The By-Clause and the WITH-Clause are removed from the representation of the retrieve request because their presence is immaterial to this discussion on concurrency control. Update requests are one of the three forms

\[ \text{UPDATE } Q \text{ } A,1 \]

or

\[ \text{UPDATE } Q \text{ } A,2 \]

or

\[ \text{UPDATE } Q \text{ } A,3 \]

The first form of the update request represents an update with a type-0 modifier, the second form represents an update with a type-I modifier and the third form represents an
update with a type-II modifier. We recall, from Chapter IV, that an update request with a modifier of type-III or IV is actually executed in MDBS as a retrieve request followed by an update request with a modifier of type-0. Hence, we do not need to consider update requests with modifiers of type-III or IV. In the three forms of the update request shown above, Q is the query used to select the records to be updated and A is the attribute being modified. If we do not wish to distinguish between the three forms of the update, we will write

```
UPDATE Q A
```

We will say that two queries Q1 and Q2 are disjoint if the records which satisfy Q1 and the records which satisfy Q2 do not have a record in common. For example (Salary>50) and (Salary<50) are disjoint. If two queries are not disjoint, they are said to overlap. We will also say that attribute A belongs to query Q, if A is also one of the attributes in the predicates of Q. Finally, we say that attribute A belongs to target-list <Y> if A is also one of the attributes in <Y>.
6.5 Request Permutabilities

Using the notation and terminology of the previous section, we will now present pairs of requests which are (could be) permutable. As we recall, a pair of requests \( r_1, r_2 \) is (could be) non-permutable if the result of executing \( r_2 \) followed by \( r_1 \) is (could be) different from the result of executing \( r_1 \) followed by \( r_2 \). We present below ten pairs of such requests. Of these, pairs (4), (7) and (10) represent pairs of requests that could be non-permutable and the remaining ones are pairs of requests that are non-permutable.

(1) INSERT \( R \)

\[ \text{RETRIEVE Q <A>, where R satisfies Q.} \]

An example of the pair is

\[ \text{INSERT (Salary,1000>, <Dept=TOY>)} \]
\[ \text{RETRIEVE (Salary>100) <Dept>} \]

(2) DELETE Q1

\[ \text{RETRIEVE Q2 <A>, where Q1 and Q2 overlap.} \]

An example of the pair is

\[ \text{DELETE (Salary>50)} \]
\[ \text{RETRIEVE (Salary>25) <Dept>} \]

(3) UPDATE Q1 A

\[ \text{RETRIEVE Q2 <A>, where Q1 and Q2 overlap} \]

and A belongs to <A> but...
not to Q2.

For example,

UPDATE (Salary>50) <Rank=25>

RETRIEVE (Salary>50) <Rank>

(4) UPDATE Q1 A

RETRIEVE Q2 <A1>, where A belongs to Q2.

For example,

UPDATE (Salary>25) <Salary=Salary-2>

RETRIEVE (Salary<25) <Rank>

(5) INSERT R

DELETE Q, where R satisfies Q.

For example,

INSERT (<Salary,1000>, <Dept,TOY>)

DELETE (Salary>50)

(6) INSERT R

UPDATE Q A where R satisfies Q.

For example,

INSERT (<Salary,1000>, <Dept,TOY>)

UPDATE (Salary>50) <Dept=SALES>

(7) UPDATE Q1 A

DELETE Q2, where A belongs to Q2.

For example,

UPDATE (Salary>25) <Salary=Salary-2>

DELETE (Salary<25)
(8) UPDATE Q1 A,2
    UPDATE Q2 A,2, where Q1 and Q2 overlap and
    the type-I modifier of one
    update consists of an addition
    or subtraction and the type-I
    modifier of the other update
    consists of some multiplication
    or division.
    For example,
    UPDATE (Salary>25) <Rank=Rank+1>
    UPDATE (Salary>50) <Rank=Rank*2>

(9) UPDATE Q1 A,1
    UPDATE Q2 A,2, where Q1 and Q2 overlap,
    and modifiers are of types 0
    and I, respectively.
    For example.
    UPDATE (Salary>25) <Rank=25>
    UPDATE (Salary>50) <Rank=Rank+1>

(10) UPDATE Q1 A
    UPDATE Q2 A,2, where A belongs to Q2.
    For example,
    UPDATE (Salary>50) <Salary=Salary+25>
    UPDATE (Salary>75) <Rank=Rank+1>
Note that there are two pairs of non-permutable requests in which one request is a retrieve and the other request is an update. These are the pairs (3) and (4). In pair (3), the retrieve request will retrieve the same set of records whether it is executed before or after the update. However, if it is executed after the update, the modifications made by the update will be visible in the retrieved records and these modifications will not be there if the retrieve request is executed before the update. In pair (4), the retrieve request will actually retrieve a different set of records if it is executed before the update than it will if it is executed after the update.

As we have said before, our concurrency control algorithm must maintain an ordering among all non-permutable requests. However, a little thought shows us that this does not apply to all cases. For example, record insertion takes place at only a single back-end. An ordering needs to be maintained only among non-permutable requests which are executed at all back-ends. Thus, an ordering needs to be maintained only among all non-permutable requests which are not inserts. To put it another way, our concurrency control algorithm will not need to check for pairs (1), (5) and (6) of non-permutable requests.
Next, let us consider the pairs of incompatible requests.

6.6 Request Compatibilities

To repeat, two requests $r_1$ and $r_2$ are compatible if their simultaneous execution gives the same result as would be obtained if we execute in the sequence $r_1$ followed by $r_2$ or in the sequence $r_2$ followed by $r_1$. Clearly, for two requests to be compatible, they must also be permutable. However, two requests may be incompatible without being non-permutable. For instance,

UPDATE (Salary>50) <Salary=Salary+2>
UPDATE (Salary>50) <Salary=Salary+2>

are incompatible but permutable. The two requests are incompatible because their simultaneous execution can lead to the problem of lost updates. However, the requests are permutable because the result of executing these updates in any order is the same.

For MDBS, it may be easily verified that the pairs (1) through (7) of non-permutable requests are also pairs of incompatible requests. Furthermore, the following three pairs of requests (8) to (10) are also incompatible.
(8) \text{UPDATE} Q_1 \ A
\text{UPDATE} Q_2 \ B, \text{ where } Q_1 \text{ and } Q_2 \text{ overlap.}
\text{For example,}
\text{UPDATE} \ (\text{Salary}>25) \ <\text{Rank}=\text{Rank}+1>
\text{UPDATE} \ (\text{Salary}>50) \ <\text{Rank}=\text{Rank}+1>

(9) \text{UPDATE} Q_1 \ A
\text{UPDATE} Q_2 \ B \text{ where } A \text{ belongs to } Q_2
\text{For example,}
\text{UPDATE} \ (\text{Salary}>25) \ <\text{Rank}=\text{Rank}+1>
\text{UPDATE} \ (\text{Rank}=20) \ <\text{Department}=17>

(10) \text{UPDATE} Q_1 \ A
\text{DELETE} Q_2, \text{ where } Q_1 \text{ and } Q_2 \text{ overlap.}
\text{For example,}
\text{UPDATE} \ (\text{Salary}>25) \ <\text{Rank}=\text{Rank}+1>
\text{DELETE} \ (\text{Salary}>50)

6.7 \text{Cluster-Based Permutabilities and Compatibilities}

In the previous sections, we described how to identify pairs of non-permutable and incompatible requests. Such identification requires, among other things, a procedure for testing if two queries overlap. Such a procedure can be quite complex. Hence, the procedures for identifying pairs of non-permutable and incompatible requests may be quite complex. In this section, we shall use the notion of
clusters to simplify the procedures for identifying non-permutable and incompatible requests.

We recall, from Chapter V, that each request in a transaction is first attached with an authorized cluster (for an insert request) or with a set of authorized clusters (for a non-insert request). In the ensuing discussion, we will represent the attached requests as follows. An attached insert request is represented by

\[ \text{INSERT } P \]

where \( P \) is the authorized cluster for the request. An attached retrieve request is represented by

\[ \text{RETRIEVE } <P> \]

where \(<P>\) is the set of authorized clusters for the retrieve request. Similarly, an attached delete request is represented by

\[ \text{DELETE } <P> \]

In the case of an update request, we term the set of future clusters as the set of all those clusters into which records may be placed after the update in accordance with the request. Then, an update request is represented by

\[ \text{UPDATE } <P> \]

where \(<P>\) is the set of clusters formed as the union of the set of authorized clusters and the set of future clusters for the update request.
Now pair (1) of non-permutable requests of Section 6.5 becomes

(a) INSERT P

RETRIEVE <X>, where P belongs to <X>.

Pair (2) of the non-permutable requests of Section 6.5 becomes

(b) DELETE <X>

RETRIEVE <Y>, where <X> and <Y> have non-null intersection.

Pairs (3) and (4) of the non-permutable requests of Section 6.5 becomes

(c) UPDATE <X>

RETRIEVE <Y>, where <X> and <Y> have non-null intersection.

Pair (5) of the non-permutable requests of Section 6.5 becomes

(d) INSERT P

DELETE <X>, where P belongs to <X>.

Pair (6) of the non-permutable requests of Section 6.5 becomes

(e) INSERT P

UPDATE <X>, where P belongs to <X>.

Pair (7) of the non-permutable requests of Section 6.5 becomes

(f) UPDATE <X>
DELETE <Y>, where <X> and <Y> have non-null intersection.

Finally, pairs (8), (9) and (10) of the non-permutable requests of Section 6.5 become

(g) UPDATE <X>

UPDATE <Y>, where <X> and <Y> have non-null intersection.

Next, let us consider the ten pairs of incompatible requests of Section 6.6. The first seven pairs of Section 6.6 are identical to those of Section 6.5 and are therefore translated into (a) through (f) as above. Pairs (8) and (9) of Section 6.6 become pair (g) above. Finally pair (10) of Section 6.6 becomes pair (f) above.

Thus, we now have a simpler decision procedure for identifying pairs of incompatible and non-permutable requests. This procedure requires us to first determine for a request the authorized cluster (for an insert request) or the set of authorized clusters (for delete and retrieve requests) or the sets of authorized and future clusters (for an update request). The algorithm for determining the authorized cluster (for an insert request) or the authorized set of clusters for (delete, retrieve and update requests) has been described in Chapter V. We shall now describe the algorithm for determining the set of future clusters for an
update request of the form

\text{UPDATE Q A}

This algorithm will utilize the set of authorized clusters already calculated.

6.7.1 Determining the Set of Future Clusters for an Update Request

Consider a record in one of the authorized clusters. We are trying to determine the cluster to which the record will belong after being updated by the request, given that we know the cluster to which the record belongs before the update. In other words, we know the set of descriptors which defines the (authorized) cluster of a record before it is updated and we also know the update request in consideration. From these two pieces of information, we are trying to determine the set of descriptors which will define the (future) cluster of the records after the record is updated.
A. Determining the Future Cluster of a Record to be Updated Without Having Seen the Record

First, we note that updates in MDBS modify only a single attribute-value. Second, we recall that the cluster to which a record belongs is defined by the set of descriptors from which every directory keyword (attribute-value) in the record is derivable. From the above observations, we conclude that the set of descriptors which defines the cluster of a record before it is updated and the set of descriptors which will define the cluster of a record after it is updated can differ in at most one descriptor. This is the descriptor from which the keyword in the record containing the attribute being modified is derivable. Therefore, we may do the following to determine the set of descriptors which will define the (future) cluster to which the updated record will belong without first examining the record. The motivation in pursuing the determination of the future cluster for a record (prior to the examination of the record) is to lock up this future cluster for subsequent retrieval and update of the record.

Consider the set of descriptors which defines the cluster to which the record belongs after update. Delete, from this set of descriptors, the descriptor from which the keyword containing the attribute being modified is
derivable. In this case, the keyword is of course a directory keyword (see Section 4.1.1). If the attribute being modified is not contained in any directory keyword, then there is no descriptor to be deleted. In either case, we must add a new descriptor to the set. This new descriptor is the one from which the keyword containing the attribute being modified will be derivable. If the modifier in the update request is of type-0, this new descriptor to be added may be easily determined, since the new value to be taken by the attribute being modified is specified in the modifier of the update request itself. However, if the modifier in the update request is not of type-0, the new value to be taken by the attribute being modified cannot be determined by examining the update request itself. Therefore, we cannot add the new descriptor to the set of descriptors. In this case, we only have a partial set of descriptors which may define several clusters each of which could potentially be used to contain the newly updated record. We will determine every such potential (future) cluster by searching the augmented CDT for clusters whose descriptor sets properly contain the partial set of descriptors at hand.
B Determining All the Future Clusters in Order to Lock Them Up For Record Updates

Since there are many records to be updated and these records are in those authorized clusters already determined, we will consider the other records to be updated with the above procedure. In other words, by repeating the above procedure for records in all the authorized clusters, we may determine the set of potential future clusters. For every cluster \( k \) in the set of potential future clusters so determined, MDBS accesses the entry in the augmented cluster definition table (CDT) corresponding to the row for cluster \( k \) and the column for the user id of the user that issued the update request. If this entry contains a field-level access control (see Chapter V) with an unspecified attribute combination part and with 'No-Insert' in the disallowed access part, MDBS will remove cluster \( k \) from the set. The above procedure is repeated for every cluster in the set. The remaining set of clusters is the set of future clusters.

More formally, let an update of the form

\[
\text{UPDATE Q A,}
\]

be issued by user \( U \). If \( A \), the attribute being modified, is not a directory attribute, the set of future clusters is identical to the set of authorized clusters. Otherwise, the set of future clusters \( F \) will have to be derived from the
set of authorized clusters P. Let the members of P be P1, P2, ..., Pk and let their corresponding descriptor sets be D1, d2, ..., Dk.

If the modifier in the update request is of type-0, extract the constant V specified in the modifier, form the attribute-value pair <A,V> and locate the descriptor D corresponding to <A,V> from the augmented DDIT. For each descriptor with A as its attribute, replace that descriptor in Di with D. If Di does not contain a descriptor with A as its attribute, add the descriptor set DDi for Di. Now use the descriptor set DDi to search the augmented cluster definition table (CDT) for clusters whose definitions include DDi. Place all such clusters into the set of future clusters F. For each cluster k in F access the entry in the augmented CDT corresponding to cluster k and user U. If the entry contains a field-level access control with an unspecified attribute combination part and with 'No-Insert' in the disallowed access part, remove cluster k from F. By repeating the above procedure for every Di, the set of future clusters F is completely determined.

Now let us describe how F may be determined when the modifier in the update request is of type-I or type-II. We do the following for each descriptor set Di, i from 1 to k. If Di contains a descriptor with A as its attribute, remove
that descriptor from the descriptor set $D_i$. If $D_i$ does not contain a descriptor with $A$ as its attribute, then do nothing to $D_i$. In either case, form the descriptor set $DD_i$ from $D_i$. Now, use the descriptor set $DD_i$ to search the augmented CDT for clusters whose definitions include $DD_i$. Place all such clusters into the set of future clusters $F$

For each cluster $k$ in $F$, access the entry in the augmented CDT corresponding to cluster $k$ and user $U$. If the entry contains a field-level access control with an unspecified attribute combination part and with 'No-Insert' in the disallowed access part, remove cluster $k$ from $F$. By repeating the above procedure for every $D_i$, the set of future clusters $F$ is completely determined.

Finally if the update request has a modifier of type-III or type-IV, it is executed as a retrieve request followed by an update request with a modifier of type-0.

C. Determining Incompatible and Non-permutable Requests

Once we have determined the sets of authorized and future clusters, the pairs (a) to (g) of incompatible and non-permutable requests may be easily determined.
6.7.2 A Case of 'Over-Determination'

The above simple procedure for identifying pairs of incompatible and non-permutable requests has been achieved at a price. This is because our procedure may sometimes identify a pair of requests as non-permutable (incompatible) even if it is actually permutable (compatible). This is because our procedure for determining the set of future clusters actually produces every potential cluster to which a record may belong after update. However, in reality the updated records cannot belong to every potential cluster so determined but to a single cluster. In a pair of requests - see (g) - the possibility of having non-null intersection with a larger number of potential clusters is higher than with a single cluster. Consequently, the pair may be considered non-permutable (incompatible) due to some non-null intersection over potential clusters of the paired requests. However, if the non-null intersection does not contain the single cluster to which the record eventually belongs, then the pair is neither non-permutable nor incompatible. Our price is therefore in over-classification of the requests. We believe nevertheless, that this price is worth paying because of the resulting simplification in the procedure for identifying pairs of incompatible and non-permutable requests.
6.8 The Cluster-Based Concurrency Control Algorithm

In the previous section, we described the procedures for identifying pairs of incompatible and non-permutable requests. In this section, we will describe the entire concurrency control algorithm which uses these procedures.

As described in Chapter V, all the requests in a transaction are partially executed until the set of authorized clusters for each request in the transaction has been determined. Each request in a transaction may then be attached with its authorized cluster, the set of authorized clusters or the set of authorized and future clusters. For simplicity, we will refer to these cluster(s) that are attached to a request as the cluster(s) needed by the request. As described in Chapter V, these attached requests are used to form a secure transaction. In the sequel, we refer to the union of all the clusters needed by the attached requests of a secure transaction as the clusters needed by a secure transaction. Further, we refer to a secure transaction as a transaction and an attached request as a request.

We will see that a transaction will need to lock various clusters during the course of its execution in one of four modes and in one of two categories. The four modes in which a transaction may lock a cluster are retrieve,
delete insert and update. The two categories in which a transaction may lock a cluster are the \textit{to-be-used category} and the \textit{being-used category}. When a transaction wishes to lock a cluster, it must specify both the mode and the category in which it wishes to lock the cluster.

6.8.1 Incompatible and Non-Permutable Locks

Consider a request \( r_1 \) in transaction \( t_1 \) and a request \( r_2 \) in transaction \( t_2 \). Let \( c \) be one of the clusters needed by \( r_1 \) and also one of the clusters needed by \( r_2 \). Then, both \( t_1 \) and \( t_2 \) will need to request locks on cluster \( c \). The lock requested by transaction \( t_1 \) on cluster \( c \) is said to be \textit{incompatible (non-permutable)} with the lock requested by transaction \( t_2 \) on the same cluster, if request \( r_1 \) of \( t_1 \) and request \( r_2 \) of \( t_2 \) form a pair of incompatible (non-permutable) requests. By considering the pairs of incompatible (non-permutable) requests of Section 6.7, we may easily derive the pairs of incompatible (non-permutable) locks shown in Figure 42. For instance, Figure 42 shows that in any category a lock on cluster \( i \) in the delete mode is incompatible and permutable with a lock on cluster \( i \) in the insert mode for all \( i \). Now, our concurrency control algorithm, instead of using the pairs of requests of Section 6.7, may use the lock table of Figure 42 to determine
<table>
<thead>
<tr>
<th></th>
<th>Delete Lock</th>
<th>Insert Lock</th>
<th>Update Lock</th>
<th>Retrieve Lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete Lock</td>
<td>C,P</td>
<td>I,P</td>
<td>I,N</td>
<td>I,N</td>
</tr>
<tr>
<td>Insert Lock</td>
<td>I,P</td>
<td>C,P</td>
<td>I,P</td>
<td>I,P</td>
</tr>
<tr>
<td>Update Lock</td>
<td>I,N</td>
<td>I,P</td>
<td>I,N</td>
<td>I,N</td>
</tr>
<tr>
<td>Retrieve Lock</td>
<td>I,N</td>
<td>I,P</td>
<td>I,N</td>
<td>C,P</td>
</tr>
</tbody>
</table>

I: Incompatible
C: Compatible
N: Non-permutable
P: Permutable

Figure 42. The Lock Table
incompatible (non-permutable) requests. From Figure 42, we see that only the modes of two locks (and not their categories) are used in deciding if they are incompatible and non-permutable. Hence, in the sequel, we can also talk of incompatible and non-permutable lock modes. We are now ready to describe the execution sequence of a transaction.

6.8.2 The Execution Sequence of a Secure Transaction

The execution sequence of a transaction begins when the transaction locks all the clusters needed by the transaction in the to-be-used category. The mode in which a cluster is locked depends on the type of request to be executed on that cluster. Thus, if cluster i is needed by a retrieve (delete, insert or update) request, it will be locked in retrieve (delete, insert or update) mode. If cluster i is needed by more than one request in the transaction, it may be locked in more than one mode. The locking of clusters by a transaction is performed independent of whether or not other transactions have locked the same clusters in any mode or category.

Now, execution of the requests of the transaction begins. Before a retrieve (delete, insert or update) request in a transaction may be executed, the clusters
needed by the request must be locked in retrieve (delete, insert or update) mode and in the being-used category. Since the clusters have been locked in some mode in the to-be-used category, the appropriate locks must be converted from the to-be-used category to the being-used category. A lock on cluster 1 may be converted only if no other incompatible lock is held on cluster 1 by another transaction in the being-used category and no other non-permutable lock is held on cluster 1 by an earlier transaction in any category. An earlier transaction is one which has arrived earlier than the present transaction at the back-end. The first check ensures inter-consistency and the second check ensures an ordering among non-permutable requests. Together, the two checks ensure monolithic consistency. After the appropriate locks have been converted, the request may be executed. Then, the locks needed by the request are released. The above procedure is repeated for every request in the transaction until all the requests in the transaction are executed.

6.8.3 The Concurrency Control Mechanism

In the actual implementation, each back-end will maintain one queue for each cluster stored at that back-end. Each element in a cluster queue (say, for cluster k) will
contain three pieces of information. First, it will contain a transaction number. Second, it contains the mode in which the transaction is holding (or wishes to hold) a lock on cluster \( k \). Third, it contains the category in which the transaction is holding (or wishes to hold) a lock on cluster \( k \).

These cluster queues will be consulted by MDBS whenever locks are requested by transactions. For instance, when a transaction locks all clusters in the to-be-used category, MDBS will create an element for each cluster needed by it and the above elements will be placed in the appropriate cluster queues. Similarly, whenever a lock conversion is requested by a transaction, MDBS consults these cluster queues. Thus, when transaction 1 requests the conversion of a lock on cluster \( k \), the system will search the queue for cluster \( k \). If an element for a transaction holding an incompatible lock in the being-used category is found, the conversion cannot be granted. Likewise, if an element for an earlier transaction holding a non-permutable lock in any category is found, the conversion cannot be granted. Whether the conversion is granted or not, the category of the queue element for the transaction is changed from to-be-used to being-used. However, if the conversion is not granted, the transaction is deactivated. That is, its execution is suspended. The deactivated transaction will
later be reactivated when the transaction holding the incompatible or non-permutable lock releases that lock. A reactivated transaction will continue execution from the point where it was suspended.

The cluster queues are also consulted whenever a transaction releases a lock on a cluster. In this case, the appropriate element will be removed from the queue for that cluster. Furthermore, the system will search the queue for that cluster to determine if any of the locks requested by other transactions may now be granted. If so, the transactions whose locks may now be granted are activated.

Three algorithms are presented here. The first algorithm is executed when a transaction is first received at a back-end and requests to lock all clusters needed by the transaction in the to-be-used category. The second algorithm is executed whenever a transaction wants to convert a lock request from the to-be-used category to the being-used category. The third and final algorithm is executed whenever a transaction releases all the locks needed by one of the requests in the transaction. We use the following notation to simplify the statement of the algorithms.

Q(i) : The queue of transactions (more precisely, transaction numbers) on cluster i.
\[ Q(i,j) : \text{The } j\text{-th element in } Q(i). \]

\[ \text{MODE}(i,j) : \text{The mode in which the } j\text{-th element in } Q(i) \text{ is holding a lock.} \]

\[ \text{CAT}(i,j) : \text{The category of the lock specified in element } Q(i,j). \]

\[ \text{TRAN}(i,j) : \text{The transaction number in } Q(i,j). \]

Let \( \text{COM}(A,B) \) imply that lock modes \( A \) and \( B \) are compatible and let \( \text{PER}(A,B) \) imply that the lock modes are permutable. Similarly, let \( \text{NOTCOM}(A,B) \) imply that lock modes \( A \) and \( B \) are incompatible and let \( \text{NOTPER}(A,B) \) imply that lock modes \( A \) and \( B \) are non-permutable. The \( j\)-th element is removed from \( Q(i) \) by "Remove \( Q(i,j) \)" and placed as the last element of the queue by "Into \( Q(i) \)". We now present these three algorithms, known as Algorithm Initialize, Algorithm Lock-Convert and Algorithm Lock-Release in Figure 43.

In the above discussion, we assume that a separate queue would be maintained for each cluster stored at a back-end. An alternative would be to consolidate all these queues into a single queue at each back-end. An element in such a consolidated queue represents a lock that a transaction holds (or wishes to hold) and must contain four pieces of information. The four pieces of information needed in a consolidated queue element are the transaction number of the transaction holding (or wishing to hold) the
ALGORITHM INITIALIZE
/* Executed when a transaction is first received at a back-end*/
/* C(i) denotes the set of clusters needed for request i in transaction T,*/
/* T = Transaction Number*/
/* Num = Number of Requests in Transaction.*/
/* Create a queue element QE with CAT = 'to be used' and TRAN = T.*/
For i=1 step 1 until Num do
begin
  Calculate the set C(i)
  Let M = mode in which the clusters in C(i) are locked
  Augment QE with MODE = M.
  For each element j in C(i) Insert QE into Q(j)
end
END ALGORITHM INITIALIZE

ALGORITHM LOCK CONVERT
/* Executed when a transaction T wants to convert locks for request R in*/
/* mode M from 'to be used' to 'being-used'*/
/* Calculate the set C(R) */
/* The set of clusters needed by R*/
For each element j of C(R)
begin
  let T be O(j,k)
  for i=1 to k-1 step 1
  begin
    If NOT COM(MODE(j,i),M) and CAT(j,i) = 'being-used' then
      deactivate T
    If NOT PER(MODE(j,i),M) and CAT(j,i) = 'to-be-used' then
      deactivate T
    end
  end
  CAT(j,k) = 'being-used'
end
END LOCK CONVERT

ALGORITHM LOCK RELEASE
/* Executed when a transaction T wants to release locks for request R in*/
/* mode M*/
/* Let C(R) be the set of clusters needed by R*/
For each element j of C(R)
begin
  let there be k elements in Q(j)
  MODE = MODE(j,1)
  CHECK = TRUE
  For i=1 to k while CHECK
  begin
    If COM(MODE(j,1),MODE) then
    begin
      activate TRAN(j,1)
      remove Q(j,1)
    end
    else CHECK = FALSE
  end
end
END LOCK RELEASE

Figure 43. Three Algorithms for Cluster Queue Management
lock, the cluster number on which the transaction holds (or wishes to hold) the lock, and the mode and category of the lock. The advantage of such a scheme over the previous scheme where several queues were maintained is that, now each back-end needs to maintain only a single queue. The disadvantage of such a scheme over the previous one is that in order to look for incompatible and non-permutable locks, every element in this consolidated queue must be searched. Compare this to the previous scheme where only the queue of elements for a specific cluster has to be searched. The disadvantage arises because the number of elements in the queue for a specific cluster is likely to be much less than the number of elements in the consolidated queue. However, the consolidated queue scheme may be preferable when the number of transactions expected in MDBS at any instant of time is not very large. In this case, the consolidated queue is not expected to be large and we would prefer to maintain a single consolidated queue rather than several cluster queues most of which will probably be empty.

The three algorithms presented in this section, together with the execution procedure presented in the previous section constitute the logic of the MDBS concurrency control mechanism which is distributed among the back-ends. The queues and the lock tables are the data structures of the mechanism.
6.9 An Examination of the Concurrency Control Mechanism

The MDBS concurrency control mechanism is applicable to both partitioned and centralized databases. It also differs from other concurrency control mechanisms in the way that locks are utilized. The MDBS algorithm uses four lock modes rather than the traditional two lock modes. By separating insert and delete locks from update locks, the MDBS concurrency control mechanism is able to support a greater degree of concurrency than would otherwise be possible.

6.9.1 New Solutions for Centralized-Database Concurrency Control

At a theoretical level, one of the contributions of this work has been in the identification of permutable and compatible requests, where the requests are issued in a high-level, query-based language. The authors of [Gard77] also identify permutable and compatible requests, but they do so for a language which is not query-based. On the other hand, the authors of [Eswa76] do consider a query-based language. However, they only identify compatible requests and they do not identify permutable requests for their query-based language. Furthermore, they do not differentiate insert and delete requests from update...
requests.

At a practical level, a method for enforcing locking when a query-based language is employed has been presented. The scheme is simpler than predicate locking [Eswa76] and is based on cluster locking. Two factors have contributed to the simplicity of our locking procedure over the procedure suggested in [Eswa76]. First, in [Eswa76], a query which requires locking must be checked against all existing queries. This can be time-consuming. On the other hand, our mechanism does not require checking of all outstanding queries. This is because a query is converted into a corresponding set of clusters and checking of the locks on these clusters is then performed. Secondly, the procedure in [Eswa76] for checking if two queries conflict requires converting the queries to disjunctive normal form. The time for such conversion can be exponential for some types of queries [Sava76]. In MDBS, the procedure for converting queries into a corresponding set of clusters does not require conversion of the queries into disjunctive normal form. The procedure for determining the set of corresponding clusters of a query which is not in disjunctive normal form is presented in Appendix G. It is clear from this procedure that the query does not need to be converted into disjunctive normal form. Therefore, the procedure is not exponential in the size of the query and,
in fact, is linear in the size of the query. For these two reasons, we believe that the MDBS concurrency control mechanism is superior to that of [Eswa76].

We note also that our concurrency control algorithm is deadlock-free. This is because a transaction can never obtain a lock unless it is confirmed that all earlier transactions will never need that lock or will need it only in a permutable mode. Hence, a transaction can only be blocked by earlier transactions and can never be blocked by subsequent transactions, thus avoiding deadlock. This is unlike the method of [Eswa76] or that of [Gard77], both of which may lead to deadlocks.

As a result, one of the disadvantages of the solutions of [Eswa76] over ours is that their solutions do not adequately cover the issue of starvation. That is, since transactions have to be backed-up and restarted in the event of deadlock, there must be enough evidence (say, a formal proof) to show that they will eventually complete. Such evidence is not provided in [Eswa76] or [Gard77]. Our system cannot suffer from starvation, since it is deadlock-free and does not require transaction restart.

If all the requests in transactions of MDBS are retrieve requests, the transactions will run without interfering with each other. Similarly, if all the requests
in transactions are inserts, or all the requests in transactions are deletes, the transactions will run without interfering with each other. In this respect, our concurrency control mechanism is better than all others which do not make this distinction between inserts, deletes and updates.

Finally, unlike the method presented in [Eswa76], the mechanism presented here does not follow the two-phase protocol [Eswa76]. A two-phase protocol is one in which a transaction must not acquire new locks after it has given up any lock. That is, it must acquire all the locks it needs in the beginning and then release these locks one by one, or it must acquire the locks one by one, as it needs them and then release them altogether in the end. A mechanism which uses the two-phase protocol will achieve a lower degree of concurrency than a mechanism such as ours where the locks are acquired and released as needed and where the protocol is not two-phase.
6.9.2 New Solutions for Partitioned-Database Concurrency Control

As far as we are aware, there are only two other published solutions to the problems of concurrency control for partitioned databases. These are the solutions of [Dew180] and that of System D which is also reported in [Dew180].

Before we can compare our solution to these two solutions, we need to develop some terminology. In these two solutions, each transaction has its own private buffer in which it does updates. The updates made by a transaction t cannot be seen by other transactions until t writes the contents of its private buffer onto the secondary memory. At this time, t is said to commit and its updates are said to become visible. Furthermore, it will be seen that these two solutions require transactions to read, update and write entities, where the entities are attribute-values. The reading, updating and writing of entities are referred to as actions of a transaction. A transaction t1 is said to come before a transaction t2 in the serialization order if

(a) t1 reads an entity which t2 writes later, or
(b) t1 writes an entity which t2 reads later, or
(c) t1 writes an entity which t2 writes later.

It is clear from the definition that the fact that t1 comes
before \( t_2 \) does not necessarily imply that \( t_2 \) does not come before \( t_1 \). Finally, a conflict is said to occur between \( t_3 \) and \( t_4 \) if \( t_3 \) comes before \( t_4 \) and \( t_4 \) comes before \( t_3 \).

Let us briefly try to describe the solutions of [Dewi80] and System D. These solutions do not employ locking for concurrency control. They wait until a transaction is ready to commit. When a transaction is ready to commit, the system will try to detect any conflicts. If no conflicts are detected, the transaction is allowed to commit. Otherwise, a number of alternative actions may have to be taken by the system. Three alternative actions are proposed in [Dewi80]. The first leads to the so-called starvation solution, the second leads to the non-starvation solution and the third leads to the restrictions-list solution. The starvation solution may cause transactions to wait perpetually as pointed out in [Dewi80]. Furthermore, we will show, by means of an example, that the starvation solution is erroneous in that it may result in an inconsistent database state. The non-starvation solution, on the other hand, causes transactions to be backed-up unnecessarily. Therefore, the restrictions-list solution of [Dewi80] is the best of the three solutions. Briefly, the restrictions-list solution is as follows. When a transaction commits, a list of prohibited actions will be immediately formed and associated with all other active
transactions in the system. These prohibited actions, which are actions that will cause a conflict, are stored in a restrictions list, one list per active transaction. When a transaction tries to perform an action in its restrictions list, the transaction is backed-up and restarted. This completes our explanation of the restrictions-list solution of [Dewi80]. The solution of System D may be considered as a distributed version of the restrictions-list solution of [Dewi80]. Having briefly explained these two solutions, we are now ready to compare the MDBS solution with these two solutions.

A. Rich Semantics in DML and New Concept of Permutability

The first weakness of these two solutions is that they use a simple data manipulation language which allows only for reading and writing entities. MDBS, on the other hand, employs a query-based data manipulation language (DML). Secondly, these systems make no effort to identify permutable requests. Because of these two weaknesses, these two systems will cause transactions to be backed-up unnecessarily. Consider the following scenario. Let there be two transactions t1 and t2. t1 increments the value of x by 2 and the value of y by 3. t2 increments the value of x by 3. That is, t1 consists of the following
Also, t2 consists of

t2: read y
read x
x=x+3
write x
commit

Furthermore, let us assume that the actions of t1 and t2 are interspersed in the following manner.

Both the restrictions-list and System D solutions would cause t2 to be backed-up after t1 commits. That is, the actions after the action, t1: commit, shown above will not be allowed by these two solutions. This is because there is a conflict between t1 and t2. Let us elaborate on this conflict. In the serialization, t2 comes before t1. This
is due to the fact that t2 has read entity y before the update of y by t1 became visible. However, t1 also comes before t2, when t2 tries to read x after t1 has written x. Hence, there is indeed the conflict. Consequently, t2 will be backed up.

With our understanding of the concept of permutability among requests, we observe that since t1 was reading x only to update it and since the two updates on x are permutable, there is no need to back-up t2. This is because the result of the execution sequence shown above is exactly the same as the result that would have been produced if t2 came before t1 in the serialization. The unnecessary back-up is caused because the two proposed solutions of [Dewi80] and System D do not differentiate between reading an entity for the purpose of transmitting the value to the user and reading an entity for the purpose of updating its value. This is due to the lack of semantics in the data manipulation language of System D. Besides the lack of semantic difference between these two kinds of reads, there is also the lack of any concept of permutable updates. For these two reasons, the proposed solutions of [Dewi80] and System D [Dewi80] failed to allow a perfectly 'legal' execution sequence to proceed.
B. Better Throughput and Lower Control Message Traffic

The MDBS concurrency control mechanism allows transactions to commit at different times in different back-ends. Consequently, at a given instant, a transaction may have committed at one back-end and not committed at another back-end. This may happen if many more records have to be accessed at one back-end than at another one in order to satisfy the requests in the transaction. We believe that this leads to increased concurrency and therefore better throughput because each back-end is executing transactions at its own pace and does not have to wait for other back-ends to complete the execution of these transactions. In other words, the execution mode is truly MIMD. On the other hand, solutions such as those of [Dewi80] and System D, which require all back-ends to commit a transaction simultaneously are essentially SIMD.

To commit a transaction simultaneously both the solution of [Dewi80] and that of System D require all the back-ends to exchange control messages. For example, in the case of [Dewi80], a special computer designated as the concurrency control computer will wait for n commit messages from the n back-ends before committing a transaction. Similarly, in the case of System D, each back-end must wait to receive (n-1) commit messages from the other (n-1)
back-ends before committing a transaction. Control messages are also needed to broadcast the restrictions list to all the back-ends.

The MDBS solution, on the other hand, exchanges no messages among the back-ends for concurrency control. This is because neither are the transactions required to commit simultaneously nor is the restrictions list employed. Hence, this solution serves to alleviate the control message traffic problem introduced in Chapter II.

C. No Back-end Limitation Problem

Another problem with the solution of [Dewi80] but not with the solution of System D is that its concurrency control mechanism is implemented in a single dedicated back-end. This can lead to an unreliable system, should the back-end fail for whatever cause. It also violates our principle of distributing all the work among all the back-ends. The MDBS concurrency control, on the other hand, does not have the back-end limitation problem.
D. A question of Overhead Incurred During Concurrency Control

It is claimed in [Dewi80] that the aforementioned two solutions incur less overhead than locking-based solutions. However, this may be debatable. First of all, for each transaction, these solutions need to keep track of which entities have been read and written by what transactions. Is this any simpler than locking the entities that have been read and written by the transactions? Furthermore, these two solutions require the maintenance of restrictions lists which incur an overhead that cannot be found in MDBS.

E. Free From Starvation Errors

We now show that the starvation solution of [Dewi80] can lead to an inconsistent database. For instance, consider two transactions t5 and t6 as follows issued to an employee database.

\begin{verbatim}
t5: read x
   y=10
   write y
   x=x+2
   write x
   commit

   t6: read y
   y=y+2
   write y
\end{verbatim}
read x
x = x * 3
write x

That is, transaction t5 increments the value of x by 2 and changes the value of y to 10. Similarly, transaction t6 increments the value of y by 2 and multiplies the value of x by 3. The corresponding transactions T5 and T6 in MDBS might be

T5: BOT
UPDATE (File=EMPLOYEE) <y=10>
UPDATE (File=EMPLOYEE) <x=x+2>
EOT

T6: BOT
UPDATE (File=EMPLOYEE) <y=y+2>
UPDATE (File=EMPLOYEE) <x=x*3>
EOT

Before these two transactions are received, the initial values of x and y in the database are assumed to be 5 and 6, respectively. Then, after the execution of these two transactions, the final values of x and y will be 21 and 12, respectively, if t5 (T5) is executed before t6 (T6). Similarly, the final values of x and y will be 17 and 10, respectively, if t6 (T6) is executed before t5 (T5). Either of these two sets of final values for x and y leaves the
database in a consistent state. Any other set of values for x and y leaves the database in an inconsistent state.

E.1 Transaction Execution by the MDBS Solution

Consider how these two transactions are executed in MDBS. Let us consider, for simplicity, that T5 was received before T6. Then it is easy to see from the description of the MDBS concurrency control algorithm in Section 6.8 that MDBS will execute these two transactions in either one of the following sequences.

Sequence one

T5: UPDATE (File=EMPLOYEE) <y=10>
T5: UPDATE (File=EMPLOYEE) <x=x+2>
T6: UPDATE (File=EMPLOYEE) <y=y+2>
T6: UPDATE (File=EMPLOYEE) <x=x*3>

Sequence two

T5: UPDATE (File=EMPLOYEE) <y=10>
T6: UPDATE (File=EMPLOYEE) <y=y+2>
T5: UPDATE (File=EMPLOYEE) <x=x+2>
T6: UPDATE (File=EMPLOYEE) <x=x*3>

and will not permit any other execution sequence. Both the permitted execution sequences above leave the final value of
x as 21 and the final value of y as 12. Therefore, the final state of the database is consistent, irrespective of which execution sequence is followed in MDBS.

E.2 Transaction Execution by the Starvation Solution

Now, let us consider how the transactions t5 and t6 are executed if the starvation solution of [Dewi80] is used for concurrency control. In that solution, the following execution sequence

\[
\begin{align*}
t5 &: \text{read } x \\
t6 &: \text{read } y \\
t5 &: y = 10 \\
t5 &: \text{write } y \\
t5 &: x = x + 2 \\
t6 &: \text{write } x \\
t5 &: \text{commit} \\
t6 &: y = y + 2 \\
t6 &: \text{write } y \\
t6 &: \text{read } x \\
t6 &: x = x + 3 \\
t6 &: \text{write } x \\
t6 &: \text{commit}
\end{align*}
\]

will be permitted except for the fact that t5 will not be allowed to commit until t6 commits. This is because t6 comes before t5 in the serialization order, since it reads an entity y which t5 will write later. After both t6 and t5 commit, the final values of x and y are 7 and 10, respectively. For the database to be consistent, the final values of x and y must be either 21 and 12 or 17 and 10.
Hence, the database is in an inconsistent state. We conclude, therefore, that the starvation solution of [Dewi80] can lead to an inconsistent database.

6.10 The Execution of Incompletely-Specified Transactions

In this section, we will consider how the MDBS concurrency control mechanism may be extended to handle incompletely specified transactions. That is, we wish to execute a transaction even before all the requests in the transaction have been provided by the user. For instance, a transaction may consist of three requests. The user has specified only one request of the transaction. We want MDBS to start executing the one request without waiting for the user to provide the remaining two requests. First, we will show why our basic mechanism is not able to execute incompletely-specified transactions. Then, we will present two possible extensions to the basic concurrency control mechanism for executing incompletely-specified transactions.

Let transaction t1 begin with the following request

RETRIEVE (File=EMPLOYEE) (Salary)

Also, let transaction t2 begin with the following request

UPDATE (File=EMPLOYEE) (Salary=Salary+2)

Furthermore, let t1 be received before t2. Since t1 is
received before t2, the retrieve request of t1 may be executed at each back-end. Now, the back-ends must decide whether or not to execute the update request of t2. The problem with executing the update request of t2 is as follows. After executing the update request of t2, MDBS may receive the second request of t1. For instance, it may be the following

```
UPDATE (File-EMPLOYEE) <Salary=Salary+2>
```

This update request of t1 is non-permutable with the update request of t2. The basic concurrency control algorithm of MDBS requires us to ensure that whenever two non-permutable requests are received, the request of the earlier transaction is executed first and the request of the later transaction is executed next. By this principle, the update of t1 should have been executed before the update of t2 is executed. However, the update of t2 has by now been already executed, whereas the execution of the update of t1 has not yet begun. Therefore, we need to back-up transaction t2. Backing-up a transaction requires the transaction to give up all its locks and to start its execution all over again.
6.10.1 Problems With Backing Up Transactions

Consider the following record in our discussion

\(<\text{Employee JAI} \>& \;<\text{Salary 5000}>\))

Two transactions, t3 and t4, increment the salary of employee JAI by 100. Consider that t3 first increments the salary to 5100 and t4 later increments it to 5200. Assume also that both t3 and t4 consist of other requests and that a back-up of t3 is required after t3 and t4 both update the salary attribute of the record and before either t3 or t4 completes its transaction. In order to back-up t3, we need to undo all the changes made by t3, thereby restoring the database to the state prior to the execution of t3. In particular, we need to restore the value of the salary attribute of the record to its original value of 5000. As a result, the update of t4 is lost. Hence, t4 must also be backed up so that its update may be executed again. In other words, backing up of one transaction may cause another transaction to be backed up. In general, backing up of one transaction may cause several other transactions to be backed up. Hence, such back-ups are costly and time-consuming.

Our second motivation against transaction back-up can be illustrated with the following scenario. Let there be two back-ends in MDBS. Let us assume that the concurrency
control at back-end 1 causes a transaction to back-up. The backed-up transaction at back-end 1 will give up its locks and start execution again from the beginning at a later time. Meanwhile, however, the same transaction is able to run to its completion at back-end 2 without any back-up. This may lead to two different (execution) sequences of the transactions' requests at the two back-ends, thus causing a loss of monolithic consistency. Thus, to be able to execute incompletely-specified transactions inMDBS, two alternatives are available to us. In one solution, we ensure that no backing up of transactions is required. In the second solution, we ensure that, even if backing-up is required, the two problems characterized above are eliminated. We will consider these two solutions in turn.

6.10.2 The No-Back-Up Solution

Before we may propose such a no-back-up solution, we repeat the cause for backing up transactions as illustrated in the previous example. There, a transaction has to be backed up because the transaction contains a request being executed which is non-permutable with a request in an earlier transaction. The occurrence of this situation is due to the fact that the earlier transaction is not completely specified and the non-permutable update of the
earlier transaction is received only after the later transaction has begun execution of its update request. The solution now becomes obvious. Before we begin to execute an incompletely specified transaction, we must ensure that all earlier transactions are completely specified. Thus, if we allow for incompletely-specified transactions in MDBS, we will execute an incompletely-specified transaction only if all earlier transactions are completely specified. This is our first solution to the problem of handling incompletely-specified transactions.

6.10.3 A Solution with Backing Up

The second solution for executing incompletely-specified transactions is as follows. We begin to execute incompletely-specified transactions even when all earlier transactions are not completely specified. This, as we know may lead to transaction back-up. There are two problems caused by transaction back-up. First, when one transaction is backed up, other transactions may also need to be backed up. This happens for the following reason. Consider that transaction t1 updates a record R1 in cluster 1. After the update, t1 releases the lock on cluster 1. This lock is subsequently acquired by transaction t2 which also updates R1 in cluster 1. Later on, t1 needs to be
backed up and the new values of attributes in R1 have to be restored to their original values prior to that update by t1. This causes the update of t2 to be lost. Hence, t2 also needs to be backed up. In other words, backing up of t1 causes backing up of t2. This happens only because t1 has released the lock on cluster 1 before t1 completes. This lock released by t1 is subsequently acquired by t2. The problem would not arise if t1 holds on to all its locks until the end of transaction. In the terminology of concurrency control, the transactions must follow the two-phase locking protocol. This takes care of the first problem associated with transaction back-up.

The second problem has to do with the fact that a transaction may be backed up at one back-end and not at another. Hence, the transactions are executed in different orders in the different back-ends and this leads to a loss of monolithic consistency. One approach is to ensure that if a transaction is backed up at one back-end, it is backed up at all back-ends. This may be achieved by exchanging control messages among the various back-ends. However, we wish to alleviate the control message traffic problem in MDBS and, hence, we reject this approach. The other approach is to ensure that even if a transaction has to be backed up at one back-end, the transaction retains the same position in the order of execution at all the back-ends.
This may be achieved by ensuring that all transactions follow the two-phase protocol and hold all their locks until the very end of execution of the transaction. Then, if they have to be backed up in the middle of execution, they still have all their locks and, hence, they will maintain their position in the order of execution.

To summarize, both problems related to transaction back-up may be overcome by ensuring that all transactions follow the two-phase protocol and hold all their locks until the very end of execution of the transaction. The process of transaction execution in MDBS when incompletely-specified transactions are allowed and when the solution with back up is adopted is as follows.

At the beginning, a transaction will lock all the clusters it needs (this set of clusters is only incompletely specified at this time) in the to-be-used category and the appropriate mode (see Section 6.8.2). Now, the transaction will have its requests executed one by one. Before a request in a transaction may be executed, the appropriate locks have to be converted from the to-be-used to the being-used category. The process of lock conversion has already been explained before. After the appropriate locks have been converted, the request may then be executed. However, after the execution of the request, the locks used
by the request are not released. They will not be considered for release until the transaction becomes completely specified and all requests in the transaction are executed. Even then, the locks are not released until all earlier transactions have been completely specified and executed and have released all their locks. Only then does this transaction release all its locks.

We now explain what happens when a new request of an incompletely-specified transaction is received. The request is executed until the set of clusters needed by the request is determined. This set is used to update the set of clusters needed by the transaction. If the new request is non-permutable with a previously executed request from a later transaction, the later transaction has to be backed up and re-executed. The back-up is achieved by resetting the values of attributes in records updated by the transaction to their original values.

This completes our description of the second solution for executing incompletely-specified transactions in MDBS. We wish to emphasize that, in the above solution, the need for transaction back-up is detected the moment a non-permutable request from an earlier transaction is received. This is in contrast to the solutions of [Dewi80] and System D where the need for transaction back-up is not
detected until the very end of a transaction. Together with the one in the previous section, we have suggested two solutions for MDBS to execute incompletely-specified transactions. Either solution may be employed in MDBS.
In Chapter I, we stated that we would propose the design of a multiple back-end system in which we would try to achieve the ideal goal of response time being inversely proportional to the multiplicity of back-ends. In the preceding five chapters, we revealed the design of such a system, known as MDBS. We now determine how well the ideal goal has been achieved by the MDBS design.

There are two possible approaches to determining how well the ideal goal has been achieved in MDBS. One may use analytical models based on queueing theory to analyze the flow of information in MDBS and to measure the designed features of MDBS. One may also use simulation techniques in order to analyze and measure the behavior of MDBS.
cting an appropriate strategy for directory management and request execution. As a result of the analytical modelling used in Chapter IV, we conclude that the ideal goal has been achieved and may even be surpassed by the MDBS design. However, a number of shortcomings of that analytical study is cited here. First, only retrieve requests were modelled and insert, delete and update requests were not modelled in that study. Second, the analytic model used is a closed queueing network model in which the total number of requests in the system is fixed. The modelling of MDBS as a closed system is valid as long as all the users of MDBS are issuing requests from terminals. However, to take care of users who may submit requests as background transactions, we would also like to model MDBS as an open system in which the total number of requests in the system is not fixed but is dependent on the arrival rate of requests and the speed of MDBS in processing these requests. This is not done in the analytic study of Chapter IV. The third shortcoming of the analytical study is that many of the finer design details of MDBS can not be modelled. For instance, the concept of clusters and the placement strategy employed to place these clusters are not modelled in the analytical study of MDBS in Chapter IV. Finally, the complexity of MDBS design and the limitations of queueing theory together render the use of
either the closed or the open queueing network model impossible for getting meaningful theoretical results on the finer design and performance details of MDBS. Thus, we decided to employ simulation techniques in order to overcome the shortcomings of the analytic study of Chapter IV.

The organization of the rest of this chapter is as follows. In Section 7.1, we propose a simulation model of MDBS. In Section 7.2, we present a measure of performance and the parameters of our simulation model. In Section 7.3, we present our results and interpretation regarding the design and performance of MDBS.

7.1 A Simulation Model of MDBS

As we know, MDBS consists of a controller attached to a number of back-ends via a time-shared bus which we shall refer to as the broadcast bus (see Figure 7 again). In our simulation model, we assume that the controller is a VAX-11/780 computer and that the back-ends are PDP-11/44 minicomputers. Furthermore, the disk drives in our simulation model are assumed to have the characteristics of the RM02 disk drive. This characterization is realistic since the proposed MDBS is to be implemented on a VAX-11/780 controller and PDP-11/44 back-ends with RM02 disk drives.
Thus, our simulation study can be used to predict the design and performance of the system to be implemented. Conversely, the result of our simulation study of MDBS design and performance can be verified by the actual performance of the implemented systems.

Our model simulates the sequence of events that takes place between the time a user request enters MDBS and the time the response data for the request is sent out of MDBS to the user. This sequence of events varies depending upon the type of the request. We describe, below the sequence of events corresponding to each of the four request types in MDBS.

7.1.1 Sequence of Events for a Retrieve Request

The processing of the request takes place in several distinct phases as described below.

A. The Parsing Phase

When a request is scheduled for execution, the VAX 11 controller first parses the request and then broadcasts the request as a message to all the back-ends. The message goes via the VAX Unibus and then via the broadcast bus to all the back-ends. It is assumed that both the VAX Unibus and the
Broadcast bus are reliable and guarantee delivery of all messages in the same order that they are presented to them. In our model, we simulate first-in-first-out (FIFO) queues at the VAX Unibus and at the broadcast bus. The time taken to transmit a message over the VAX Unibus or the broadcast bus depends, naturally, upon the size of the message.

B. The Descriptor Search Phase

The broadcasted retrieve request is eventually placed in a FIFO queue at each of the back-ends. Each back-end processes its queue sequentially. On encountering this request in its queue, each back-end will perform descriptor processing for it. That is, if there are \( x \) predicates in the query in the request, and there are \( n \) back-ends, each back-end will process \( x/n \) of the predicates in the query to determine the corresponding descriptors of the predicates. In determining the corresponding descriptors, a back-end may need to access the secondary memory because the augmented descriptor-to-descriptor-id-table (DDIT) which stores the descriptors may be in the secondary memory. In that case, an I/O request is generated by the back-end and placed in the queue associated with the disk drive which contains the augmented DDIT. Thus, in addition to a FIFO queue for user requests, the back-end has a number of I/O queues.
Each disk drive has an I/O queue of access operations which have been issued by the back-end to which the disk drive is attached. The disk drives are assumed to use a first-come-first-served (FCFS) strategy in processing access operations in their queues. When a seek to a disk track is completed by a disk drive, the disk drive must wait until the Unibus of the PDP-11/44 back-end to which it is attached becomes free before it may transmit the selected track to the back-end.

Eventually, the back-end will complete descriptor processing on the request. It will then broadcast the corresponding set of descriptors so determined to all the back-ends via the broadcast bus. For a request of n predicates, a back-end must complete processing x/n of the predicates and receive the corresponding set of descriptors for the remaining predicates of the query in the request from the other (n-1) back-ends before it can do any further processing on that request. However, in the meantime, the back-end may process other requests in its fifo queue.

C. The Address Generation Phase Including Access Control

Consider that a back-end has finished the descriptor processing phase for a request and that it has received the remaining (n-1) corresponding sets of descriptors for this
request from the other \((n-1)\) back-ends. The back-end may then proceed to do address generation for that request by accessing and searching its augmented cluster definition table (CDT). At the end of this phase of processing, the addresses (of the records) of the authorized clusters for the user request have been determined by the back-end. Originally, these addresses were generated for each cluster at the database-creation time in accordance with the track-splitting-with-random-placement strategy for placing the records of the cluster in the secondary storage.

D. The Secondary Memory Retrieval Phase

Finally, the back-ends have to access the tracks containing the records of the permitted set of clusters. Thus, the back-ends will generate a set of I/O requests, one for each track to be accessed, and place these I/O requests in the queues of the appropriate drives. As the back-end receives tracks of records from the drives, it processes these tracks by selecting those records which satisfy the user's query. Thus, the processing of tracks of records at the back-ends proceeds in parallel with the accessing of other tracks by the disk drives.
E. The Response Phase

Values of target attributes are extracted from the qualifying records and sent over the broadcast bus and via the VAX-11 Unibus to the VAX-1 controller. When the entire response to the request has been received from all the back-ends, the controller will output the response set back to the user.

7.1.2 Sequence of Events for a Delete Request

The first four phases of a delete sequence are exactly the same as the ones for a retrieve sequence. However, the following phases are different.

A. The Tag-for-Deletion Phase

The selected records are marked with deletion tags. The marked records are then written back to the secondary store. In order for a back-end to write records with deletion tags onto the secondary store, the back-end must generate some I/O operations and place them in the fcfs queues of appropriate disk drives.
B The Acknowledgement Phase

After the records have been properly written to secondary store, a message is sent to the controller, via the broadcast bus and the VAX-11 Unibus, indicating the successful completion of the delete request. When such acknowledgement has been received from all the back-ends, the controller outputs a positive acknowledgement to the user.

7.1.3 Sequence of Events for an Update Request

As we recall, an update request in MDBS is associated with a modifier of type-0, I, II, III or IV. An update request with a modifier of type-III or IV is processed as a retrieve request followed by an update request of type-0. Thus, we simulate only update requests with modifiers of types-0, I and II.

The first four phases of an update sequence are exactly the same as the ones of a retrieve sequence. However, the remaining two phases are different.
A. The Record Modification and Cluster Calculation Phase

Each back-end must update the selected records by employing the modifier in the update request and calculate a new cluster number for each updated record. In order to keep the simulation of updates simple, we assumed that updated records did not change clusters. Rather, they continued to belong to the same cluster(s) and are inserted back into the same track(s) from which they are retrieved. Then, the updated records have to be written back to the secondary memory.

B. The Acknowledgement Phase

After the update is successfully completed, the back-end sends a message to the controller, via the broadcast bus and the VAX-11 Unibus indicating completion of the update request. When such messages have been received from all the back-ends, the controller will output a positive acknowledgement to the user.

7.1.4 The Sequence of Events for an Insert Request

As for the other three request types, the execution of an insert request also proceeds in several distinct phases.
A. Parsing Phase

When an insert request is scheduled for execution, the VAX-11 controller will parse the request and then broadcast it to all the back-ends via the VAX Unibus and the broadcast bus.

B. The Descriptor Search and Initial Address Generation Phases

The request is now placed in the FIFO queues at the n back-ends. When a back-end encounters this request in its queue, the back-end will find the corresponding descriptors for x/n of the total of x directory keywords in the record for insertion. In order to find the corresponding descriptors, a back-end has to search the augmented DDIT. If the necessary descriptors are in the secondary memory an I/O operation is generated by the back-end and placed in the fcfs queue associated with the disk drive which contains the descriptors in question.

After a back-end finds the corresponding descriptors for x/n of the directory keywords, the back-end will then broadcast its corresponding descriptors to all the other back-ends via the broadcast bus. Each back-end must complete the processing of the x/n directory keywords and receive the corresponding descriptors for the remaining
(n-1)x/n directory keywords from the other (n-1) back-ends. Only then, can the back-end do further processing on the insert request. However, the back-end may process other requests in its queue while it is waiting for the corresponding descriptors to be broadcasted from the other back-ends.

After completing the descriptor processing on the insert request and receiving the (n-1) messages from the other back-ends, the back-end proceeds to the first step of address generation. In this step, it searches the augmented CDT to determine the cluster number of the record for insertion. Then, in the second step of address generation, it checks to see if the user that issued the insert request is allowed to make the insertion into the cluster determined in the first step of address generation. If not, the execution of the request terminates at this point. Otherwise, the back-end sends the cluster number to the controller via the broadcast bus and the VAX-11 Unibus.

C. The Back-end Selection Phase

The controller waits for messages from all the n back-ends. Then, it consults the cluster-id-to-next-back-end-table (CINBT) for selecting a next back-end for the record insertion. The controller will
then send a message to this back-end via the VAX-11 Unibus and the broadcast bus. The message is placed in the FIFO queue at the selected back-end.

D. Record Insertion Phase

On encountering the insertion message in its queue, the back-end proceeds with the final step of address generation. It will first search the augmented CDT to determine the track into which the record is to be inserted and then generate an I/O request to the disk drive containing the selected track. After the back-end receives a message from the disk drive informing it of the successful insertion of the record, the back-end will, in its turn, send a message to the controller via the broadcast bus and the VAX-11 Unibus.

E. The Acknowledgement Phase

Upon receiving a positive message from a single back-end, the controller acknowledges to the user that the record is successfully inserted.
7.2 Simulation Environments and A Measure of Performance

The simulation model described in the last section is programmed using Simula on a DEC System 20. Because the model is highly parameterized, the number of distinct cases that can be formulated is extremely large. The combinatorics inherent to such modelling makes any exhaustive simulation infeasible. Fortunately, a great deal can be learned by simulating an appropriately chosen subset of all the cases involved. See appendices H and I.

7.2.1 Retrieve-Intensive vs. Update-Intensive

Let the environment of MDBS in a particular application be defined by the percentages of the four request types that will be encountered in that application. Then, a retrieve-intensive environment is one in which a large percentage of requests received by MDBS are retrieve requests and in which very few delete, update and insert requests are received. Similarly, an update-intensive environment is one in which a large percentage of requests received by MDBS are update, delete and insert requests, and in which the percentage of retrieve requests encountered is low. Any of the large possible set of environments may be accommodated by our simulation model. However, we choose to model two specific environments — a retrieve-intensive one
in which 100% of the requests are retrieves and an update-intensive one in which there will be 25% of each of the four request types. Thus, we are modelling the two ends of the spectrum of possibilities for the environment of MDBS.

7.2.2 Cluster Size vs. Request Size

In our study we assumed that a total of 10000 clusters exist in the database and that these clusters are placed using the track-splitting-with-random-placement strategy described in Chapter II. The average number of tracks in a cluster is chosen from the set [2,10]. In the case of update, delete and retrieve requests, we also need to choose the number of clusters that will be updated, deleted or retrieved by a request. We run tests for small-sized requests in which the number of clusters to be processed (updated, deleted or retrieved) varies between 1 and 20, and for large-sized requests in which the number of clusters to be processed varies between 20 and 40. The number of predicates in a query is chosen to be uniformly distributed in the range from 1 to 5.
7.2.3 Hardware Configurations and Requirements

The broadcast bus is chosen to have a transmission speed of 1 Mbytes/sec and the VAX Un bus is chosen to have a transmission speed of 2 Mbytes/sec. It is assumed that both the controller and the back-ends have to execute about 8000 instructions, taking 8 msecs, in order to generate messages for broadcast. Furthermore, we assume that each message has to be augmented with 50 bytes of header information. In other words, the minimum possible size of a message in MDBS is 50 bytes. Finally, the number of back-ends is chosen from the set \([3, 6, 9]\).

7.2.4 A Measure of Anticipated Performance vs. Ideal Performance

In order to be able to present our results more effectively, we define a parameter called the percentage ideal goal. We take the response time of MDBS with three back-ends as our reference point. Then, we define the percentage ideal goal of MDBS utilizing \(n\) back-ends as:

\[
\frac{3 \times \text{(response time of MDBS with 3 back-ends)} \times 100}{n \times \text{(response time of MDBS with } n \text{ back-ends)}}
\]

By the above definition, we see that the percentage ideal goal of MDBS utilizing six back-ends will be 100, only if the response time of MDBS with six back-ends is exactly half
the response time of MDBS with three back-ends. Similarly, when the number of back-ends is nine, the percentage ideal goal of MDBS will be 100 only if the response time of MDBS with nine back-ends is exactly one-third the response time of MDBS with three back-ends. Thus, the percentage ideal goal of MDBS is a measure of how close MDBS is to achieving its ideal performance goal.

7.3 MDBS Performance Under Various Conditions

On the basis of the measure, environments and parameters, we conduct the simulation of MDBS. In this section, we tabulate the simulation results and discuss these tables. In the discussion, we try to interpret the findings and relate the findings to the design details of MDBS.

7.3.1 Intensive Retrieval involving Large Clusters

From Figure 44, we see that the performance of MDBS is better than anticipated when the average number of tracks per cluster is chosen to be ten. In fact, the percentage ideal goal gets to be as high as 140 for large-sized requests when the number of back-ends is six and the
### Small-sized Requests (involving 1 to 20 clusters)

<table>
<thead>
<tr>
<th>back-ends</th>
<th>3</th>
<th></th>
<th>6</th>
<th></th>
<th>9</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Response time (sec)</td>
<td>% Ideal Goal</td>
<td>Response time (sec)</td>
<td>% Ideal Goal</td>
<td>Response time (sec)</td>
<td>% Ideal Goal</td>
</tr>
<tr>
<td>1.5</td>
<td>2.06</td>
<td>100</td>
<td>.84</td>
<td>122.6</td>
<td>.583</td>
<td>117.78</td>
</tr>
<tr>
<td>2</td>
<td>1.06</td>
<td>100</td>
<td>.786</td>
<td>105.6</td>
<td>.567</td>
<td>97.6</td>
</tr>
</tbody>
</table>

### Large-sized Requests (involving 20 to 40 clusters)

<table>
<thead>
<tr>
<th>back-ends</th>
<th>3</th>
<th></th>
<th>6</th>
<th></th>
<th>9</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Response time (sec)</td>
<td>% Ideal Goal</td>
<td>Response time (sec)</td>
<td>% Ideal Goal</td>
<td>Response time (sec)</td>
<td>% Ideal Goal</td>
</tr>
<tr>
<td>3.5</td>
<td>4.53</td>
<td>100</td>
<td>1.66</td>
<td>136.44</td>
<td>1.1</td>
<td>140.3</td>
</tr>
<tr>
<td>5</td>
<td>3.16</td>
<td>100</td>
<td>1.51</td>
<td>104.64</td>
<td>1.05</td>
<td>100.3</td>
</tr>
</tbody>
</table>

Average Number of Tracks Per Cluster is 10

Figure 44. The MDBS Response Times In a Retrieve Intensive Environment with Large Amount of Data Involved
interarrival time of requests is 3.5 secs. The reason for
the percentage ideal goal being greater than 100 has been
explained in Chapter IV and has to do with the utilization
of the disk system. To repeat, when the number of back-ends
in MDBS is doubled, the number of tracks to be retrieved by
each back-end is halved because of our data placement
strategy. Furthermore, the time to access a track from disk
secondary memory is less when more back-ends are used
because the utilization of the disk subsystem is lower in
this case. Both these factors together contribute to a
response time improvement which is better than ideal.

7.3.2 Intensive Retrieval Involving Small Clusters

From Figure 45, we also see that the percentage ideal
goal can be as low as 61.5 when the average number of tracks
per cluster is two and the requests are small-sized.
Similarly, the percentage ideal goal can get as low as 70
when the average number of tracks per cluster is two and the
requests are large-sized. In other words, the performance
of MDBS deviates furthest from the ideal goal when the
average number of tracks per cluster and the average number
of clusters to be retrieved by a request are both very
small. The interpretation is as follows. Our design has
strived to alleviate the controller limitation problem and
### Small-sized Requests (involving 1 to 20 clusters)

<table>
<thead>
<tr>
<th>back-ends</th>
<th>3</th>
<th>6</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Response time in sec</td>
<td>% Ideal Goal</td>
<td>Response time in sec</td>
<td>% Ideal Goal</td>
</tr>
<tr>
<td>1.5</td>
<td>.571</td>
<td>100</td>
<td>.364</td>
</tr>
<tr>
<td>2.0</td>
<td>.55</td>
<td>100</td>
<td>.359</td>
</tr>
</tbody>
</table>

### Large-sized Requests (involving 20 to 40 clusters)

<table>
<thead>
<tr>
<th>back-ends</th>
<th>3</th>
<th>6</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Response time in sec</td>
<td>% Ideal Goal</td>
<td>Response time in sec</td>
<td>% Ideal Goal</td>
</tr>
<tr>
<td>1.5</td>
<td>1.04</td>
<td>100</td>
<td>.586</td>
</tr>
<tr>
<td>2.0</td>
<td>.957</td>
<td>100</td>
<td>.567</td>
</tr>
</tbody>
</table>

The average number of tracks per cluster is 2.
Retrieve-intensive environment where all requests are retrieve requests.

Figure 45. The MDBS Response Times in a Retrieve-Intensive Environment with Small Amount of Data Involvement
the control message traffic problem. However, neither of these two problems could be entirely eliminated in our design. Thus, a constant amount of time, independent of the number of back-ends, is spent by the controller to parse user requests, to broadcast user requests to all back-ends and to output results to the user. Similarly, the back-ends need to exchange messages in order to synchronize descriptor processing on a request. The effects of these overhead tasks on the response time of a request, and, hence, on the achievement of the ideal goal will be negligible when the number of tracks to be retrieved from secondary memory is large. This is because when a large fraction of the response time will be spent in accessing secondary memory, the effects of these tasks on the overall response time is relatively low. On the other hand, if the number of tracks to be retrieved from secondary memory and processed is very small, then the overall response time to a request is also very small. In that case, the time for performing the overhead tasks will be a much larger proportion of the overall response time. This is why the performance of MDBS deviates furthest from the ideal goal when the average number of tracks in a cluster and the average number of clusters to be retrieved for a request are both very low.
7.3.3 Intensive Update Involving Large Clusters

The results tabulated in Figure 46 for an update-intensive environment are similar to those of Figure 44. Thus, when the average number of tracks per cluster is ten, the percentage ideal goal can get as large as 190. This happens for large-sized requests when the number of back-ends is nine and the inter-arrival time of requests is 3.5 sec.

In comparing Figure 46 with Figure 44, we note that the highest percentages achieved are 190 and 140, respectively. One may wonder why the update-intensive requests achieve higher percentage (i.e., 190) than the retrieve-intensive requests under the same conditions, since updates tend to tax the system performance more pronouncedly than retrieves do. It turns out that these two percentages are not directly related to each other. What they have indicated is that with more back-ends, the slow update (with a response time of over six seconds) in a 3-back-ends setting may be speeded up more dramatically than the fast retrieval (with a response time under five seconds). By the time MDBS is a system of 9 back-ends, the response times of updates and retrievals are both close to one second as depicted in Figures 46 and 44, respectively.
### Small-sized Requests (1 to 20 clusters)

<table>
<thead>
<tr>
<th>back-ends</th>
<th>3</th>
<th>6</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Response time in sec</td>
<td>% Ideal Goal</td>
<td>Response time in sec</td>
</tr>
<tr>
<td>1.5</td>
<td>3.1</td>
<td>100</td>
<td>1.03</td>
</tr>
<tr>
<td>2.0</td>
<td>2.24</td>
<td>100</td>
<td>.946</td>
</tr>
</tbody>
</table>

### Large-Sized Requests (20 to 40 clusters)

<table>
<thead>
<tr>
<th>back-ends</th>
<th>3</th>
<th>6</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Response time in sec</td>
<td>% Ideal Goal</td>
<td>Response time in sec</td>
</tr>
<tr>
<td>3.5</td>
<td>6.73</td>
<td>100</td>
<td>2.24</td>
</tr>
<tr>
<td>5.0</td>
<td>5.67</td>
<td>100</td>
<td>2.08</td>
</tr>
</tbody>
</table>

Average Number of Tracks Per Cluster is 10.
The Update-intensive Environment Consists of 25% Inserts, 25% Deletes, 25% Updates and 25% Retrieves.

Figure 46. The MDBS Response Times in an Update-Intensive Environment Involving Large Clusters
7.3.4 Intensive Update Involving Small Clusters

For small values of the average number of tracks in a cluster and the average number of clusters to be accessed in an update request, the percentage ideal goal can get as low as 60. Comparing corresponding entries in Figures 45 and 47, we see that the performance of MDBS is closer to ideal in the update-intensive environment than in the retrieve-intensive environment.

The reason for this is explained below. Update and delete requests take longer to execute than retrieve requests because, besides requiring the retrieval of tracks from secondary memory, the former require the insertion of tracks containing the updated and deleted records to secondary memory. As a result, the average response time of MDBS in an update-intensive environment is greater than in a retrieve-intensive environment. Hence, the overall response time in an update-intensive environment will be less affected by the time for the fixed 'overhead' tasks such as request parsing and acknowledgement than the overall response time in a retrieve-intensive environment.
### Small-sized Requests (1 to 20 clusters)

<table>
<thead>
<tr>
<th>back-ends</th>
<th>3</th>
<th>6</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Response time in sec</td>
<td>% Ideal Goal</td>
<td>Response time in sec</td>
<td>% Ideal Goal</td>
</tr>
<tr>
<td>1.5</td>
<td>.604</td>
<td>100</td>
<td>.398</td>
</tr>
<tr>
<td>2.0</td>
<td>.581</td>
<td>100</td>
<td>.391</td>
</tr>
</tbody>
</table>

### Large-sized Requests (20 to 40 clusters)

<table>
<thead>
<tr>
<th>back-ends</th>
<th>3</th>
<th>6</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Response time in sec</td>
<td>% Ideal Goal</td>
<td>Response time in sec</td>
<td>% Ideal Goal</td>
</tr>
<tr>
<td>1.5</td>
<td>1530</td>
<td>100</td>
<td>.759</td>
</tr>
<tr>
<td>2.0</td>
<td>1260</td>
<td>100</td>
<td>.702</td>
</tr>
</tbody>
</table>

The Average Number of Tracks Per Cluster is 2.
The update-intensive environment where there are 25% of each of the four request types.

Figure 47. Intensive Update Involving Small Clusters
7.3.5 Effects of Broadcasting on Performance

It is easy to see that for a given request, the amount of data which is returned to the controller from the back-ends to be output to the user is a constant, independent of the number of back-ends. As a result, the time taken to transmit this data over the broadcast bus is a part of the overall response time of MDBS which cannot be decreased by increasing the number of back-ends. Thus, a slow broadcast bus may cause the performance of MDBS to deviate away from the ideal goal. In this section, we propose to investigate the effects of broadcast bus speed on the performance of MDBS.

In the previous section, we presented results on MDBS response time assuming that the broadcast bus could transmit at a rate of 1 Mbyte/sec. We now present two more sets of results, one for the case when the broadcast bus can transmit at 0.5 Mbytes/sec and the other for the case when the broadcast bus can transmit at 2 Mbytes/sec. Since the deviation of MDBS performance away from ideal is greatest for small number of tracks per cluster (2) and small-sized requests, we present results only for this case. Furthermore, since the effects of a slow bus will be felt the most in a retrieve-intensive environment where large amounts of data have to be returned over the broadcast bus,
we will only simulate such an environment. The results are tabulated in Figure 48.

The results indicate that as long as the broadcast bus can transmit at a speed greater than 0.5 Mbytes/sec, the overall response time of MDBS is unaffected. Note that these results should be read in conjunction with the results in Figure 45 for the case when the broadcast bus can transmit at a speed of 1 Mbyte/sec. We conclude that the speed of the broadcast bus is not likely to affect the performance of MDBS in its achievement of the ideal goal, since buses with speeds of greater than 0.5 Mbytes/sec are commercially available. An example of one such bus is the PCL-ll bus [Uhri79] provided by Digital Equipment Corporation.

7.3.6 Three Observations of Strong Design and Performance Factors — High-Volume Processing, Intensive Update and Inexpensive Broadcast Bus

Let us summarize our results up to this point. MDBS will achieve and surpass its ideal goal as long as the requests issued to it are such that large amounts of data have to be read and manipulated in order to process them. This is precisely the kind of environment for which MDBS has
### Bus speed = 0.5 Mbytes/sec

<table>
<thead>
<tr>
<th>back-ends</th>
<th>3</th>
<th>6</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.5</td>
<td>.571</td>
<td>100</td>
<td>.364</td>
</tr>
<tr>
<td>2.0</td>
<td>.55</td>
<td>100</td>
<td>.359</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Bus speed = 2 Mbytes/sec

<table>
<thead>
<tr>
<th>back-ends</th>
<th>3</th>
<th>6</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.5</td>
<td>.571</td>
<td>100</td>
<td>.364</td>
</tr>
<tr>
<td>2.0</td>
<td>.55</td>
<td>100</td>
<td>.359</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Average Number of Tracks Per Cluster is 2.
The environment is Retrieve-intensive with small-sized requests (1 to 20 clusters)

Figure 48. The Response Times of MDBS Effected by the Broadcast Bus Speeds
been designed. On the other hand, the percentage ideal goal of MDBS may be as low as 60 if requests are such that only very small amounts of data have to be read and manipulated in order to process them.

Secondly, our results have shown us that MDBS will achieve its ideal goal more closely in an update-intensive environment than in a retrieve-intensive environment because of the additional accesses to secondary memory needed in the former case.

Finally, our results have shown us that the speed of the broadcast bus is not a bottleneck to its ideal performance. As long as the speed of this bus is greater than 0.5 Mbytes/sec, the performance of MDBS is unaffected by the speed of the broadcast bus.

7.4 A More Refined Simulation of MDBS

In the simulation experiments of the previous section, we had assumed that the controller broadcasts a request to all the n back-ends in MDBS. Actually, the controller will need to broadcast a request to only x back-ends, where x is the number of predicates in a retrieve, delete or update request or the number of keywords in an insert request, if x is less than n. By use of such a policy, the
synchronization overhead in MDBS is reduced. Thus, during the descriptor processing phase, each back-end will need to wait for results from only \((x-1)\) rather than \((n-1)\) other back-ends. Furthermore, such a policy will result in fewer message exchanges and lesser traffic on the broadcast bus. This refined policy is now incorporated into the simulation model of MDBS.

The response time results for MDBS using the refined policy are tabulated in Figure 49. The corresponding results for MDBS without the refined policy are the ones in Figure 45. We only simulate MDBS with the refined policy when the average number of tracks per cluster is small, i.e., two. This is because the performance of MDBS is ideal or better when the average number of tracks per cluster is ten even without use of the more refined policy. Thus, we are only interested in seeing if the refined policy can improve the performance of MDBS in the region where it is operating below ideal. Comparing corresponding entries in Figures 45 and 49, it is clear that MDBS response times under the more refined policy are lower than those without such a policy. However, more importantly, the MDBS performance is now much closer to ideal. Thus, in the worst case, the percentage ideal goal of MDBS is 86 as opposed to 60 when the refined policy is not employed. We do not believe that MDBS performance can be any closer to ideal.
The average number of tracks per cluster is 2
The environment is Retrieve-intensive

### Small-sized Requests (1 to 20 clusters)

<table>
<thead>
<tr>
<th>back-ends</th>
<th>3</th>
<th>6</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Response time in sec</td>
<td>% Ideal Goal</td>
<td>Response time in sec</td>
</tr>
<tr>
<td>1.5</td>
<td>0.521</td>
<td>100</td>
<td>0.286</td>
</tr>
<tr>
<td>2.0</td>
<td>0.507</td>
<td>100</td>
<td>0.283</td>
</tr>
</tbody>
</table>

### Large-sized Requests (20 to 40 clusters)

<table>
<thead>
<tr>
<th>back-ends</th>
<th>3</th>
<th>6</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Response time in sec</td>
<td>% Ideal Goal</td>
<td>Response time in sec</td>
</tr>
<tr>
<td>1.5</td>
<td>0.104</td>
<td>100</td>
<td>0.466</td>
</tr>
<tr>
<td>2.0</td>
<td>0.915</td>
<td>100</td>
<td>0.454</td>
</tr>
</tbody>
</table>

Figure 49. The Response Times ofMDBS Under a Refined Policy Simulation
because the controller limitation problem and the control message traffic problem can never be completely eliminated.
CHAPTER VIII

DESIGN GOALS AND ACHIEVEMENTS - A SUMMARY AND REVIEW

It is generally known that the use of a single general-purpose digital computer with dedicated software for database management as a back-end to offload the mainframe host computer from database management tasks yields no appreciable gains in performance. Furthermore, to replace the back-end computer hardware and software with more powerful hardware and newly designed software may be costly and disruptive. Such an upgrade would require major effort, since these single back-end database management systems are not designed for hardware extension and software enhancement.

In this dissertation, we first make the claim that for the management of very large databases the use of multiple minicomputers in a parallel fashion may be feasible and desirable. By feasible we mean that it is possible to configure a number of back-end minicomputers each of which
is driven by identical database management software and controlled by a controller minicomputer for concurrent operations on the database spread over the disk storage local to the back-end minicomputers. This approach to large databases may be desirable because only off-the-shelf equipment of the same kind is utilized to achieve high performance without requiring specially-built hardware and because identical database management software may be replicated on the back-ends. This approach makes the capacity growth and performance improvement easy because duplicate hardware can be added and used with replicable software.

We then present, in this dissertation, a new approach to the solution of database management problems involving database growth and performance improvement. This approach utilizes a multiplicity of conventional minicomputers, novel hardware configuration and innovative software design. This extensible system, called MDBS, is designed to achieve the ideal goal of having the response time be inversely proportional to the multiplicity of back-ends.

In presenting our approach, our initial effort was to identify the problems and bottlenecks involved in designing such an ideal system. These problems were identified by surveying typical software-oriented multiple back-end
systems in existence. In order to overcome the problems identified and in order to develop an ideal system, we set nine design goals for MDBS.

First, we resolve to eliminate the channel limitation problem. That is, we do not wish the throughput of MDBS to be limited by the transfer rate of data from secondary store via I/O channels. Second, we resolve to alleviate the controller limitation problem by executing all the database management functions in a parallel fashion in the back-ends rather than at the controller. For our third design goal, we decide that all back-ends must execute identical software. As a result, capacity growth and performance improvement by use of additional back-ends containing the replicable software is simplified. Fourth, we resolve to minimize communications among the back-ends and between the back-ends and the controller. Without excessive communications, the performance of MDBS will not taper off after the first few additional back-ends. Fifth, we decide not to use any special-purpose hardware in MDBS. This makes the addition of duplicate hardware for capacity growth and performance improvement easy. It also makes the replication of software on additional hardware easy. Sixth, we decide that each back-end should support concurrent request execution for better resource utilization and system response time. For our seventh goal, we resolve that more
than one disk drive would be attached to each back-end. As a result, very large databases of the magnitude of, say, $10^{10}$ bytes can be supported in MDBS. Eighth, we decide to design MDBS in such a way that all back-ends would participate equally in the execution of a request. This is essential for achieving an ideal system in which response time is inversely proportional to the number of back-ends.

Finally, for our ninth goal, we resolve to find a canonical data model into which prevailing data models such as the relational, network and hierarchical models can be translated in a straightforward manner.

We began our design of MDBS by specifying that it would consist of a controller and several back-ends. Each back-end would have attached to it several disk drives which would store the database. As a result of attaching several disk drives to each back-end, we are able to achieve our seventh goal. Furthermore, we are also able to overcome the channel limitation problem and, hence achieve our first goal. Thus, the throughput of MDBS is no longer limited by the transfer rate of data from the secondary store via I/O channels. By using off-the-shelf equipment for the controller, the back-ends and the disk drives, we are able to achieve our fifth goal of not using any special-purpose hardware in MDBS. Thus, we achieve three of our nine design goals.
Our second goal is to ensure that none of the major database management tasks are done in the controller but are done in a parallel fashion by the back-ends. This is achieved by the careful design of algorithms for request execution. Before MDBS may access the database in order to execute a request, it must first access auxiliary information about the database. That is, MDBS must perform directory management for the request. A number of alternative strategies for directory management are proposed and evaluated by using a closed queueing network model. The new strategy chosen for directory management minimizes overall response time and requires minimal work to be performed at the controller of MDBS. Next, during the execution of a request, MDBS enforces access control. That is, MDBS must perform access-control-related directory management. The proposed scheme for enforcing access-control is also performed at the back-ends and is not done in the controller. In order to improve the response time of a request even further, it is necessary to have a mechanism which allows for concurrent execution of multiple requests. Such a mechanism is referred to as a concurrency control mechanism. The newly proposed concurrency control mechanism for MDBS also executes in the back-ends and not at the controller. As a result, every phase of request execution is performed parallelly in the back-ends and
Our second goal is to ensure that none of the major database management tasks are done in the controller but are done in a parallel fashion by the back-ends. This is achieved by the careful design of algorithms for request execution. Before MDBS may access the database in order to execute a request, it must first access auxiliary information about the database. That is, MDBS must perform directory management for the request. A number of alternative strategies for directory management are proposed and evaluated by using a closed queueing network model. The new strategy chosen for directory management minimizes overall response time and requires minimal work to be performed at the controller of MDBS. Next, during the execution of a request, MDBS enforces access control. That is, MDBS must perform access-control-related directory management. The proposed scheme for enforcing access-control is also performed at the back-ends and is not done in the controller. In order to improve the response time of a request even further, it is necessary to have a mechanism which allows for concurrent execution of multiple requests. Such a mechanism is referred to as a concurrency control mechanism. The newly proposed concurrency control mechanism for MDBS also executes in the back-ends and not at the controller. As a result, every phase of request execution is performed parallelly in the back-ends and
requires minimal work at the controller. Thus, we manage to alleviate the controller limitation problem and achieve our second goal in MDBS.

Each of the algorithms mentioned above for request execution requires that identical software be executed in the back-ends. As a result, we achieve the third goal of MDBS to have replicable software in the back-ends to facilitate capacity growth and performance improvement by use of additional back-ends.

Our fourth goal is to minimize communications among the back-ends and between the back-ends and the controller. This is achieved by a number of different techniques. First, the controller and the back-ends are connected by means of a broadcast bus. As a result, the controller will need to use only a single message rather than n different messages in order to send a request to all the n back-ends. Simulation experiments show that the reduced message traffic caused by use of the broadcast capability can lead to improvements in response time by a factor of as much as five. Communications traffic is also reduced by ensuring that the concurrency control mechanism in MDBS requires no exchange of messages among the back-ends. This is unlike all other concurrency control mechanisms for multiple back-ends that we are aware of. Finally the communications
traffic is reduced by the use of a data placement strategy which ensures that each back-end has to retrieve the same amount of data as any other back-end in order to respond to a request. Thus, a back-end does not need to communicate with other back-ends or with the controller in order to retrieve data from the database. These techniques allow us to achieve the fourth goal of alleviating the control message traffic problem in MDBS.

Our sixth goal is to ensure that each back-end in MDBS support concurrent request execution for better resource utilization and system response time. We first argue that such concurrent request execution is beneficial for MDBS. Next, we determine the necessary and sufficient conditions for a consistent MDBS database that utilizes multiple back-ends. As a result, we develop the very important notion of monolithic consistency. A new algorithm for ensuring monolithic consistency is then described. Our algorithm is unique in a number of ways. First, it advocates the use of four lock modes instead of the traditional two lock modes. By separating the insert and delete locks from the update locks, we achieve a greater degree of concurrency. Another contribution is the identification of permutable and compatible requests for a high-level query language such as MDBS' DML. At a practical level, a method for enforcing locking when using a
predicate-based query language is proposed. Unlike [Eswa76] which uses predicate locking, our scheme uses cluster locking and is simpler. Unlike [Jord81], the scheme allows predicate-based updates. Unlike both [Eswa76] and [Jord81], the scheme is deadlock-free. Hence it cannot suffer from the 'starvation' problem where transactions are rolled back infinitely and are not guaranteed to complete. Thus, we achieve the sixth goal of MDBS.

Our eighth goal is to ensure that each back-end participates equally in the execution of a request. This is achieved by partitioning the database into equivalence classes which are termed clusters. Every record in the database belongs to one and only one cluster. In order to form clusters, the database-creator must specify descriptors. By proper use of descriptors, the clusters are formed in such a way that if a user needs to access a record belonging to a cluster, the user is most likely to have the need to access all the other records belonging to that cluster. Thus, clusters serve as the basic units of access in MDBS. In other words, every user request requires the retrieval of one or more clusters. By storing the clusters in such a way that each cluster is evenly distributed among the back-ends, we may ensure that every user request will require the retrieval of the same amount of data from all the back-ends. Thus, the record clustering and cluster
placement algorithms ensure that each back-end does an equal share of data retrieval for a request. The directory management, access control and concurrency control algorithms are also designed so that each back-end performs an equal share of the work. Thus, we have achieved our eighth goal in MDBS.

Finally, for our ninth goal, we resolve to find a canonical data model into which prevailing data models such as the relational, network and hierarchical data models can be translated in a straightforward manner. In our quest for a canonical data model, we evaluated a number of data models on the basis of three criteria - the translation criterion, the partition criterion and the language criterion. We were able to show that the attribute-based model was the only one that satisfied all three criteria. Accordingly, we choose to implement the attribute-based model directly and the other data models by translation, in MDBS. We also present a simple data manipulation language based on this data model. Thus, we have achieved our ninth and final goal for MDBS.

In this way, every one of the nine design goals set for MDBS are achieved. We believe that the resulting architecture comes close to achieving the ideal goal of having the response time be inversely proportional to the
number of back-ends. In order to test our conjecture, a complete simulation model of MDBS was designed on a DEC System 20 using Simula. Several simulation experiments are run on this model.

The results of the simulation experiments prove to be very satisfactory. It is seen that MDBS response time is ideal or better under typically expected conditions when the number of tracks to be retrieved and manipulated in order to satisfy a user request is large. The reason for the MDBS response time being better than ideal in some cases needs some explanation and is closely related to the utilization of the disks which store the database. When the number of back-ends in MDBS is increased, not only is the number of tracks to be retrieved at each back-end decreased, so is the time to retrieve each track. This explains the reason for the better-than-expected decrease in response time.

The results of the simulation experiments also show that MDBS response time is less than ideal when the number of tracks to be retrieved and manipulated is small. However, the deviation from ideal is very slight and is never more than 20%. The reason for the less-than-ideal performance is because the controller limitation problem and the problem of control message traffic cannot be entirely eliminated but can only be alleviated. More specifically,
there are four tasks which are to be executed in MDBS where the execution time of these tasks is independent of the number of back-ends in MDBS. We present these four tasks below.

First, there is the parsing task performed in the controller. That is, the controller has to spend a certain amount of time performing the parsing of a user request and this time is independent of the number of back-ends in MDBS. Second, there is the broadcasting task which is also performed by the controller and requires an amount of time independent of the number of back-ends in MDBS. For this task, the controller broadcasts a request to all the back-ends. Third, there is the outputting task. That is, the controller has to spend a certain amount of time, once again independent of the number of back-ends, in outputting the results of a request to the user that issued the request. The final task is the address generation task which is performed at each back-end. The execution of this task has been carefully designed so that each back-end has to perform less work when the number of back-ends is increased. However, under ideal conditions, a back-end must perform no address generation if none of the clusters to be retrieved are stored at that back-end. This is not achieved in our system. Thus, even when a back-end does not contain any of the clusters to be retrieved, it must still do
address generation in order to discover that it does not contain any of the clusters to be retrieved for a request.

Our simulation indicates that due to the fixed overhead of these four tasks the ideal goal set for MDBS is not being achieved when the number of tracks to be accessed and processed is very small. Nevertheless, we do not believe that the latter three tasks may be performed in any different way to improve the performance of MDBS. In other words, the latter three tasks are inherent to multiple back-end systems and are not a defect of MDBS. However, the performance of the first task may be improved, if we can come up with a parallel algorithm for parsing a user request. However, unless the user request is complex, the need for a parallel parsing algorithm for performance enhancement will be unnecessary. On the other hand, for requests involving large amounts of data, the fixed overhead incurred from the aforementioned tasks becomes negligible.

In conclusion, we believe that this dissertation has met its objectives in the design and analysis of a multiple back-end database management system for capacity growth, performance improvement and functionality enhancement.
LIST OF REFERENCES


Chu, W. W., Lee, D. and Iffla, B., "A Distributed Processing System for Naval Data


[Epst78] Epstein, R., Stonebraker, M., and Wong, E.,
"Distributed Query Processing in a Relational Data Base System," Memorandum No. UCB/ERL M78/18, Electronics Research Laboratory, University of California, Berkeley, April 1978.


[Jord81] Jordan, J. J., Banerjee, J., and Batman, R.,


[Mary77] Maryanski, F.J., "Performance of Multi-processor


[Rive76] Rivest, R. L., "Partial-Match Retrieval


Wah, B. W., and Yao, B. S., "DIALOG - A Distributed Processor Organization for Database Machine," Proceedings of the National Computer

APPENDIX A

FORMAL SPECIFICATION OF DML

The following is the BNF syntax for the DML of Chapter III. Square brackets are used to indicate optional constructs.

Predicate ::= (attribute rel-op value)
attribute ::= char-string
a-b-m ::= attribute
base-attribute ::= attribute
rel-op ::= ![<]>|>!=!#
value ::= string|number|float
Conjunct ::= (Predicate)! Conjunct & Predicate)
Query ::= (Conjunct)! (Query V Conjunct)
Stat ::= AVG!MAX!MIN!SUM!COUNT
list-el ::= Stat(attribute)
list ::= attribute!list-el!list,
        attribute!list, list-el
Target-list ::= (list)
Attrib-val-pair ::= <attribute,value>
Half-record ::= Attrib-val-pair!Half-record,
                 Attrib-val-pair
Record ::= (Half-record)
Pointer ::= number
Modifier ::= type-0!type-I!type-II!
           type-III!type-IV
type-0 ::= <a-b-m=value>
type-I ::= <a-b-m=expr1>
type-II ::= <a-b-m=expr2>
type-III ::= <a-b-m=expr2 of Query>
type-IV ::= <a-b-m=expr2 of Pointer>
Request ::= Insert!Delete!Update!Retrieve
Insert ::= INSERT Record
Delete := DELETE Query
Update := UPDATE Query Modifier
Retrieve := RETRIEVE Query Target-list
            [BY Attribute] [WITH Pointer]
uc-letter := A!B!C!...!Z
string := uc-letter!uc-letter string
lc-letter := a!b!c!...!z
char-string := uc-letter!char-string
            lc-letter
digit := 0!1!2!3!4!5!6!7!8!9
number := digit!digit number
float := number.number
add-op := +!-
mult-op := *!/
expr1 := a-term1!expr1 add-op a-term1
a-term1 := a-factor1!a-term1 mult-op
         a-factor1
a-factor1 := a-b-m!number
expr2 := a-term2!expr2 add-op a-term2
a-term2 := a-factor2!a-term2 mult-op
         a-factor2
a-factor2 := base-attribute!number
APPENDIX B

PROOF THAT A RECORD BELONGS TO ONE AND ONLY ONE CLUSTER

Consider a record R consisting of the directory keywords K1, K2, ..., Kn, where Kn contains directory attribute Ai. If n is zero, then the record R contains no directory keywords. Hence it is derived from the cluster defined by the empty set of descriptors and this is the only cluster it is derived from. Next, let us consider the case when n is non-zero. For purposes of contradiction, let a record R belong to two different clusters X and Y. Consider directory keyword K1 in R. Since R belongs to cluster X, X must contain a descriptor X1 from which K1 is derived. Similarly, X must contain a descriptor X2 from which K2 is derived. Therefore, X is defined by the descriptor set {X1, X2, ..., Xn}. Similarly, Y is defined by the descriptor set {Y1, Y2, ..., Yn}. Now, by the definition of the Xis and
the Yis, the following n statements must be true.

(1) \(K_1\) is derived from both \(X_1\) and \(Y_1\).
(2) \(K_2\) is derived from both \(X_2\) and \(Y_2\).

\[\vdots\]

(n) \(K_n\) is derived from both \(X_n\) and \(Y_n\).

Using the statement (1) above, we shall show that \(X_1 \equiv Y_1\).

The first point to be noted is that \(X_1\) and \(Y_1\) must both contain \(A_1\), since, \(K_1\) contains \(A_1\). Let \(K_1\) be of the form \(<A_1, V>\). Now, there are three possibilities for \(X_1\) as below.

(a) \(X_1\) is a type-A descriptor of the form \((L_1 \leq A \leq U_1)\).
(b) \(X_1\) is a type-B descriptor of the form \((A_1 = V_1)\).
(c) \(X_1\) is a type-C descriptor of the form \(A_1\).

Similarly, there are three possibilities for \(Y_1\) as below.

(a) \(Y_1\) is a type-A descriptor of the form \((L_2 \leq A \leq U_2)\).
(b) \(Y_1\) is a type-B descriptor of the form \((A_1 = V_2)\).
(c) \(Y_1\) is a type-C descriptor of the form \(A_1\).

Thus, there are nine possibilities for the combination of \(X_1\)
and Y₁. We shall consider all of the nine cases, in turn.

**Case 1: X₁ is (L₁ ≤ A₁ ≤ U₁).
Y₁ is (L₂ ≤ A₁ ≤ U₂).**

Now, since K₁ is derived from X₁, V lies between L₁ and U₁. Also, since K₁ is derived from Y₁, V lies between L₂ and U₂. Rule 1 of the rules for forming descriptors (see Chapter III) states that the ranges specified in type-A descriptors for a given attribute must be mutually exclusive. Since the ranges of X₁ and Y₁ are not mutually exclusive, they cannot be two different descriptors. Hence, they must be the same descriptor. Thus, X₁ = Y₁.

**Case 2: X₁ is (L₁ ≤ A₁ ≤ U₁)
Y₁ is (A₁ = V₂).**

Since K₁ is derived from X₁, V₁ lies between L₁ and U₁. Since K₁ is derived from Y₁, V₁ must be equal to V₂. Thus, V₂ lies between L₁ and U₁. However, this is in violation of Rule 2 for forming descriptors, since, there is a type-B descriptor whose value is enclosed in the range of a type-A descriptor. Hence, it is not possible that X₁ is of type-A and Y₁ is of type-B.
Case 3: \( X_1 \) is \((L_1 \leq A_1 \leq U_1)\).
\( Y_1 \) is \( A_1 \).
This is a clear violation of Rule 3 for forming descriptors which states that an attribute that appears in a type-C descriptor cannot also appear in a type-A or type-B descriptor. Thus, this case is not possible.

Case 4: \( X_1 \) is \((A_1 = V_1)\).
\( Y_1 \) is \((L_2 \leq A_1 \leq U_2)\).
Since \( K_1 \) is derived from \( X_1 \), \( V \) must be equal to \( V_1 \).
However, since \( K_1 \) is also derived from \( Y_1 \), \( V \), and hence \( V_1 \), must lie between \( L_2 \) and \( U_2 \). This is a violation of Rule 2 for forming descriptors. Hence, this case is not possible.

Case 5: \( X_1 \) is \((A_1 = V_1)\).
\( Y_1 \) is \((A_1 = V_2)\).
Since \( K_1 \) is derived from \( X_1 \), \( V \) must be equal to \( V_1 \).
But, since \( K_1 \) is also derived from \( Y_1 \), \( V \) must also be equal to \( V_2 \). Hence, \( V_1 = V_2 \). Hence, \( X_1 = Y_1 \).

Case 6: \( X_1 \) is \((A_1 = V_1)\)
\( Y_1 \) is \( A_1 \).
This is a clear violation of Rule 3 for forming
descriptors since an attribute that appears in a type-C descriptor cannot also appear in a type-A descriptor. Hence, this case is not possible.

Case 7: X is $L\leq A \leq U$.

Y is $(L_2 \leq A_1 \leq U_2)$.
Once again, we have a violation of Rule 3 for forming descriptors and, hence, this case is not possible.

Case 8: X is $A_1$

Y is $(A_1 = V_2)$
Once more, we have a violation of Rule 3 for forming descriptors. Hence, this case is not possible.

Case 9: X is $A_1$.

Y is $A_1$.
Clearly, $X_1 = Y_1$.

Out of all the nine cases considered, only three were found to be possible. In each of these three cases, X was found to be equal to Y. Thus, $X_1 = Y_1$. 
We may similarly use statement (2) to show that \( X_2 = Y_2 \), statement (3) to show that \( X_3 = Y_3 \), and so on. Thus, \( X - Y \). Hence, \( R \) does not belong to two different clusters. Hence, a record can belong to one and only one cluster.
APPENDIX C

DIRECTORY MANAGEMENT ALGORITHMS

Here, we will describe two algorithms. The first algorithm determines the cluster to which a record for insertion belongs and it also determines the secondary memory addresses of this cluster. This algorithm is called the directory management algorithm for an insert request. The algorithm is presented as two different procedures because it proceeds in two distinct phases called the descriptor search phase and the address generation phase.

The second algorithm describes how the corresponding set of clusters (and their addresses) for the query in a non-insert request are determined. This algorithm is called the directory management algorithm for a non-insert request. Once again, the algorithm is presented as two different procedures.
ble is called the pointer table (PT). It has as many entries as there are directory attributes. Each entry in the PT consists of two fields. The first field contains a directory attribute and the second field contains a pointer to another table called the descriptor-to-descriptor-id table (DDIT). The pointer in the second field of an entry in PT points to the location in the DDIT where the descriptors of the attribute in the first field of the PT entry are stored.

The DDIT has as many entries as there are descriptors. Each entry consists of two fields. The first field contains a descriptor and the second field contains a descriptor id. A view of a DDIT is shown in Figure 14.

Finally, there is a table called the cluster definition table (CDT). There are as many entries in the CDT as there are clusters in the database. Each entry in the CDT consists of three fields. These contain a cluster number, a cluster definition (as a descriptor set) and one or more secondary memory addresses, respectively. The secondary memory addresses in the third field are of the cluster in the first field. A view of a CDT is shown in Figure 15.
Physically, we assume that the CDT is implemented as follows. Consider a database with 1 directory attributes and D descriptors per attribute. Then, the CDT consists of iD entries. Each entry is formed with a descriptor id followed by a list of cluster definitions and their corresponding secondary memory addresses.

1. Directory Management for an Insert Request

We present two procedures below. The first one is executed during the descriptors search phase and the second one is executed during the address generation phase.

PROCEDURE DESCRIPTOR SEARCH

Purpose: Given a record for insertion, it determines the corresponding descriptor for each attribute-value pair in the record.

Input: The record to be inserted. Let there be n attribute-value pairs in the record for insertion. Thus, the record for insertion is

\((<A_1,V_1>, <A_2,V_2>, \ldots <A_n,V_n>)\)

Output: A set of k descriptors, where k is the number of directory attributes. These descriptors are output in the array \(D(1), D(2), \ldots, D(k)\).

Step 1: Set \(i=1\). Set \(j=1\).

Step 2: Consider the attribute-value pair \(<A_i, V_i>\). Follow the pointer in PT, \(P_i\), to the location in DT where the descriptors of \(A_i\) are stored. Let \(m=P_i\), set \(P(i+1)\) to the next pointer in PT after \(P_i\). If no pointer \(P_i\) is found, go to Step 4.

Step 3: Check if \(<A_i,V_i>\) can be derived from the \(m\)-th descriptor of DDIT. If so, set \(D(j) = m\)-th descriptor of DT, \(j=j+1\), and go to Step 4. Else, go to Step 5.

Step 4: \(i=i+1\). If \(i>n\), then go to Step 6. Else, go to Step 2.
Step 5: Set \( m = m + 1 \). If \( m < P(i+1) \), go to Step 3. Else, set \( D(j) \) = null descriptor on \( A_i \), \( j = j + 1 \) and go to Step 4.

Step 6: Output \( D(1), D(2), \ldots, D(k) \). Terminate.

PROCEDURE ADDRESS GENERATE

Purpose: Generates the cluster to which the record for insertion belongs and its secondary memory address or addresses.

Input: \( D(1), D(2), \ldots, D(k) \) which was output from the previous algorithm.

Output: The cluster to which the record belongs and the secondary memory addresses, of this cluster.

Step 1: Consider the non-null descriptors in \( D(1), D(2), \ldots, D(k) \) and let these be \( ND_1, ND_2, \ldots, ND_p \). Let the attributes in the non-null descriptors be \( AT_1, AT_2, \ldots, AT_p \). Let there be \( d_i \) descriptors in the database for attribute \( AT_i \). Choose that attribute, \( AT_j \), in \( AT_1, AT_2, \ldots, AT_p \) with the maximum number of descriptors on it, \( d_j \).

Step 2: Access the CDT entry for attribute \( AT_j \) and descriptor \( ND_j \). Search this entry until the cluster defined by \( D(1), D(2), \ldots, D(k) \) is encountered.

Step 3: Output the cluster number and its corresponding secondary addresses as stored in the entry. Terminate.

2. Directory Management for a Non-Insert Request

Once again, we shall present two procedures. The first one is executed by MDBS during the descriptors search phase and the second one is executed during the address generation phase.
PROCEDURE DESCRIPTOR SEARCH

Purpose: Given a non-insert request with a query, it determines the corresponding descriptor for each predicate in the query.

Input: The query in disjunctive normal form. Let there be i conjunctions and let there be Pj predicates in the jth conjunction.

Output: A two-dimensional array D in which D(x,y) contains the corresponding descriptor for the y-th predicate of the x-th conjunction.

Step 1: Set m=1. [At any point, the m-th conjunction is being examined]. Set n=1. [At any point, the n-th predicate of the m-th conjunction is being examined]. Set k=1. Let the array D be initially empty.

Step 2: Let the attribute in the n-th predicate of the m-th conjunction be A and let its value be V. Use PT to determine the location in DDIT where the descriptors corresponding to A are stored. Set P and START to this value. Set END to the location in DDIT where the descriptors corresponding to A end.

Step 3: Check if <A,V> can be derived from the p-th descriptor in DDIT. If so, set D(m,n) to the p-th descriptor of DDIT and go to Step 4. Else, set p=p+1. If P<END, go to Step 3. Else, go to Step 4.

Step 4: Set n=n+1. If n>Pm, then go to Step 5. Else, go to Step 2.

Step 5: Set m=m+1. If m>i, then stop. Else, set n=1 and go to Step 2.

PROCEDURE ADDRESS GENERATE

Purpose: Generates the clusters which will satisfy a user request and their secondary memory addresses.

Input: The two-dimensional array D from the previous algorithm and the user query of i conjunctions.

Output: The clusters which will satisfy the user request and their secondary memory addresses.

Step 1: Set m=1 [At any time, we examine the m-th conjunction]. Let query conjunction m have Pm predicates.
Step 2: Set $m = P_m$. Consider the descriptors $D(m,1)$, $D(m,2)$, ..., $D(m,n)$. Let the attributes in these descriptors be $AT_1$, $AT_2$, ..., $AT_n$. Also, let $D(m,i)$, $1 \leq i \leq n$, be the $x_i$-th descriptor on $AT_i$. Also, let there be $d_i$ descriptors in all, in the database, on $AT_i$.

Step 3: Associate a count $C_i$ with each $AT_i$ as follows. If the $i$-th predicate of the $m$-th conjunction contains the operator '=$', $C_i = 1/d_i$. If the $i$-th predicate of the $m$-th conjunction contains the operator '<' or '<', then set $C_i = x_i/d_i$. Else, set $C_i = (d_i - x_i)/d_i$.

Step 4: Choose the $AT_i$ with minimum $C_i$. Access the entry in the CDT for attribute $AT_i$ and descriptor $D(m,i)$. If the $i$-th predicate of the $m$-th conjunction is '<' or '<', then also search the entries corresponding to attribute $AT_i$ and all descriptors 'less than' $D(m,i)$. Else, if the $i$-th predicate of the $m$-th conjunction is '>' or '>', then also search the entries corresponding to attribute $AT_i$ and all descriptors 'greater than' $D(m,i)$.

Step 5: For each cluster encountered in searching these entries, do Step 6.

Step 6: Set $p = 1$. Consider attribute $AT_p$ and the $p$-th predicate of the $i$-th conjunction. If the predicate is '=$', then check to see if the cluster definition includes descriptor $D(m,p)$. If the predicate is '<', '<', or '<', then check to see if the cluster definition includes descriptor $D(m,p)$ or any descriptor 'less than' $D(m,p)$. Otherwise, check to see if the cluster definition includes descriptor $D(m,p)$ or any descriptor 'greater than' $D(m,p)$. If the checks fail at any point, then the cluster being examined does not satisfy the user request. Otherwise, set $p = p + 1$. If $p > n$, then output the cluster being examined and its corresponding address (or addresses). Else, go to Step 6.

Step 7: Set $m = m + 1$. If $m > 1$, terminate. Else, go to Step 2.
APPENDIX D

ALGORITHMS FOR BINARY SEARCH OF DESCRIPTORS

In this appendix, we shall calculate the time for a binary search of N descriptors. Let

tpr: time taken to perform an arithmetic operation

td: time taken to read a descriptor

N: number of descriptors

K: descriptor being searched for

g(a): the nearest integer greater than or equal to a.

h(a): the nearest integer less than or equal to a.

\( \lg \): logarithm of the base 2

Let the N descriptors we are searching be K1, K2, ..., KN. Then, the algorithm for binary search of these N descriptors is as follows.

1. \( i = g(N/2), \ m = g(N/2) \)

2. If \( K < K_i \) go to 3
   If \( K > K_i \) go to 4
   If \( K = K_i \) success
3. If $m=0$ failure
   else $i=i-g(m/2)$
   $m=h(m/2)$
   go to 2

4. If $m=0$ failure
   else $i=i+g(m/2)$
   $m=h(m/2)$
   go to 2

From the above, the total time to do binary search on $N$ descriptors is

$$2tpr + \lg(N(td + 4tpr))$$
APPENDIX E

I/O SUBMODEL FOR DISK SYSTEM

The I/O submodel consisting of M drive queues and one channel queue is shown in Figure 20. The inter-arrival distribution is chosen as exponential with $L$ as the mean rate at which requests are received by the disk system. The inter-arrival distribution of requests to each drive queue is also exponential with a mean request rate of $L/M$. Let us analyze the drive queues and the channel queue, in turn.

Analyzing a Drive Queue - The drive queue service time is the seek time. The seek time of a disk drive is approximated by an equation of the form $(a + by)$, where $a$ and $b$ are constants and $y$ is the number of cylinders traversed during the seek. We let $N$ be the total number of cylinders in a disk drive. Then,

$$\text{Mean service time} = \text{Mean seek time}$$
\[ = \text{sig}(y, 1, N)((a + by)(2/N - 2y/N^2)) \]
\[ = 2a - (2a/N^2)\text{sig}(y, 1, N)(y) \]
\[ + (2b/N)\text{sig}(y, 1, N)(y) \]
\[ - (2b/N^2)\text{sig}(y, 1, N)(y^2) \]
\[ = a - a/N + bN/3 - b/3N \]

**Second moment of service time**

\[ = \text{sig}(y, 1, N)((a^2 + by^2 + 2aby) \]
\[ (2/N - 2y/N^2)) \]
\[ = a^2 - a^2/2N + (bN)^2/6 \]
\[ - b^2/6 + 2abN/3 - 2ab/(3N) \]

**Variance of service time**

\[ = \text{Second moment of service time} \]
\[ - (\text{Mean service time})^2 \]
\[ = (bN)^2/2/18 + b^2/18 - a^2/N^2 \]
\[ - b^2/(9(N^2)) - 2ab/(3(N^2)) \]
\[ + a^2/N + 2ab/3 \]

**Analyzing the Channel Queue** — It is easy to see that

**Mean service time** = disk rotation time = d

**Variance of service time** = 0

**Mean Inter-arrival time** = 1/L

**Variance of Inter-arrival time** = 1/L^2
The channel queue is analyzed as an M/G/1 queue [Klei75] whose waiting time is

\[ L(d^{*2})/(2(1-Ld)) \]

Let total time in the channel system be the sum of the channel wait and service times. Let

- \( \text{what} \) = channel wait time
- \( \text{xhat} \) = channel service time
- \( \text{shat} \) = mean time in channel
- \( \text{w2hat} \) = second moment of channel wait time
- \( \text{x2hat} \) = second moment of channel service time
- \( \text{s2hat} \) = second moment of total time in channel
- \( \text{vars} \) = \( s2hat - (shat)^{-2} \) = variance of time in channel

Then,

\[ \text{shat} = \text{what} + \text{xhat} = L(d^{*2})/(2(1-Ld)) + d \]
\[ \text{s2hat} = \text{w2hat} + \text{x2hat} + 2(\text{xhat})(\text{what}) \]
\[ = \text{w2hat} + d^{*2} + L(d^{*3})/(1-Ld) \ldots \ldots (1) \]
\[ \text{w2hat} = (L^{*2})(d^{*4})/(2(1-Ld)^{*2}) \]
\[ + L(d^{*3})/(3(1-Ld)) \]

Substituting \( \text{w2hat} \) in (1),

\[ \text{s2hat} = (L^{*2})(d^{*4})/(2(1-Ld)^{*2}) \]
Finally,

\[ \text{vars} = s2\hat{a}t - (\hat{a}t)^2 = \frac{(L^2)(d^4)}{4(1-Ld)^2} + \frac{L(d^3)}{3(1-Ld)} \]
APPENDIX F

ADDRESS GENERATION TIMES

The time taken for address generation is really the time taken to search the augmented CDT for the addresses of the corresponding set of clusters. We assume that the augmented CDT is physically implemented as follows. Consider a database with \( i \) directory attributes and \( D \) descriptors per attribute. Then, the augmented CDT consists of \( ID \) entries. Each entry is formed with a descriptor id followed by a list of cluster definitions and their corresponding secondary addresses. Let

\[
\begin{align*}
1 & : \text{number of directory attributes} \\
D & : \text{number of descriptors per attribute} \\
t & : \text{number of bytes needed to store a track address} \\
p & : \text{number of bytes needed to indicate a back-end number} \\
c & : \text{size of a cluster in tracks}
\end{align*}
\]

463
\( \lg : \text{logarithm to base 2} \)

\( u(x) : \text{nearest integer greater than or equal to } x \)

\( n : \text{number of back-ends} \)

\( s : \text{track size} \)

\( z : \text{time to access a track from secondary memory} \)

\( x : \text{size of a CDT entry in the centralized strategy} \)

\( \text{adgen} : \text{time for address generation in the centralized strategy} \)

\( y : \text{size of a CDT entry in the other strategies} \)

\( \text{adgenl} : \text{time for address generation in the other strategies} \)

The size \( x \), of each CDT entry in the centralized strategy is

\[ u(\lg D/8) + D^*(i-1)(u(\lg D/8)i + c(t+p)) \]

So,

\[ \text{adgen} = \frac{xz}{s} \]

Similarly, the size \( y \) of each CDT entry in the remaining strategies is

\[ u(\lg D/8) + c(D**(i-1))/n(u(\lg D/8)i + t), \text{ if } c \leq n; \]

\[ u(\lg D/8) + D**(i-1)(u(\lg D/8)i + ct/n), \text{ otherwise} \]

Finally,

\[ \text{adgenl} = \frac{yz}{s} \]
APPENDIX G

DETERMINING CLUSTERS CORRESPONDING TO AN ARBITRARY QUERY

In this appendix, we will describe the algorithm for determining the set of clusters corresponding to an arbitrary query. This algorithm employs the other algorithm described in Chapter IV for determining the set of clusters corresponding to a query in disjunctive normal form. For simplicity, we will refer to the algorithm for determining the set of clusters corresponding to an arbitrary query as Algorithm Query. Similarly, we will refer to the algorithm for determining the set of clusters corresponding to a query in disjunctive normal form as Algorithm Disjunct. Algorithm Query takes an arbitrary query and generates the corresponding set of clusters for the query as output. Algorithm Disjunct takes a query in disjunctive normal form as input and generates the corresponding set of clusters for the query as output. Thus, both algorithms will have two arguments. The first argument is the input query. The
Algorithm \text{Query} \\
/*Q is the input query*/ \\
/*D is the set of clusters output for Q*/ \\
Read next character X from Q \\
Q'=unread portion of Q; \\
If X='(' then \\
begin \\
Call \text{Query}(Q',D_1); \\
Read next character of Q; \\
end \\
else \\
begin \\
Read the next predicate P of Q; \\
Call \text{Disjunct}(P,D_1); \\
end \\
again: \\
Read next character X from Q; \\
Q'=remaining portion of Q until first unread 'V' or '&'; \\
If X='V' then \\
begin \\
Call \text{Query}(Q',D_2); \\
D=D_1 \cup D_2; \\
end \\
else if X='&' then \\
begin \\
Call \text{Query}(Q',D_2); \\
D=D_1 \cap D_2; \\
end \\
else stop; \\
Go to again; \\
End \text{Query}
APPENDIX H

DESCRIPTION OF THE SIMULATION MODEL

Our simulation model of MDBS follows the so-called scenario approach [Fran77] to simulation. Thus, MDBS is simulated as a number of scenarios or processes. Consider an MDBS system with $n$ back-ends and $m$ disk drives. Then, MDBS is simulated by $n+m+3$ processes. These are the processes for the $n$ back-ends, the $m$ disk drives, the controller, the broadcast bus and the Vax Unibus. Each process in the simulation model is associated with a queue. The queue associated with a back-end is referred to as a back-end queue; the queue associated with a disk drive is referred to as a disk drive queue; the queue associated with the controller is referred to as the controller queue; the queue associated with the broadcast bus is referred to as the broadcast bus queue and the queue associated with the Vax Unibus is referred to as the Vax Unibus queue.
An element in one of these queues may be of one of several types. For instance, the controller queue contains three types of elements. First, some elements are requests which have to be parsed, processed, and so on. In the simulation model, these are elements of type 1. Second, the controller queue contains elements which are responses to requests received from the back-ends. In our model, these elements are of type 6. Finally, it contains type-4 elements. These are messages sent from the back-ends to the controller to indicate the cluster number of a record for insertion.

A back-end queue contains three types of elements. First, it contains elements which are requests on which directory processing has to be performed by the back-end. These are type-2 elements. Second, it contains elements which have been broadcast from other back-ends after directory processing. That is, after a back-end performs directory processing on a request, it broadcasts the corresponding descriptors to all the remaining back-ends. The corresponding descriptors are stored as type-5 elements in the back-end queues. Third, a back-end queue will contain elements of type 3. These are the messages broadcast by the controller after consultation of the cluster-id-to-next-back-end-table (CINBT) for an insert
request.

A disk drive queue consists only of a single type of element. These elements are I/O requests generated by the back-end to which the disk drive is attached.

The broadcast bus queue consists of elements of five types. First, it consists of requests which are initially broadcast from the controller to the back-ends via the broadcast bus. These are elements of type 2. Second, it contains type-3 elements representing messages from the controller to the back-ends after consultation of the CINBT for an insert request. Third, it contains type-5 elements made of the corresponding descriptors for a request which are broadcast from a back-end to all other (n-1) back-ends. Fourth, it contains type 6 elements which are the responses to requests sent from the back-ends to the controller. Finally, it contains type 4 elements consisting of messages from the back-ends to the controller indicating the cluster for insertion.

The Vax Unibus queue contains elements of six different types. It contains elements of types 2, 3, 6 and 4 as described above for the broadcast bus queue. In addition, it consists of type-1 elements representing requests initially submitted to MDBS. It also consists of type-7 elements, i.e. the outputs of MDBS.
DESCRIPTION OF PROCESSES

We are now ready to describe the \( n+m+3 \) processes which cumulatively describe the simulation model of MDBS. Since the \( n \) processes which model the \( n \) different back-ends are identical, and since the \( m \) processes which model the \( m \) different disk drives are identical, we only need to describe five processes. These five processes will be described below. These processes will utilize the various queues described in the previous section.

Basically, the following is done in each process. The next element from the queue associated with the process is examined. The kind of service to be performed on that element will depend on the type of the element. Having determined the type of the element, the appropriate service is performed for that element by advancing the simulation clock (i.e., delaying, in simulation terminology) for the time needed to perform that service. The service may also include placing the element in another queue. After performing the appropriate service, the element is removed from this queue and the next element in this queue is examined. The above procedure is repeated until the simulation termination criterion is met. In our model, the termination criterion is met when the simulation clock has advanced to a very large value. This completes the basic
description of each process. What varies from process to process is the kind of service which needs to be performed on the elements in the appropriate queues associated with these processes. Let us now describe each of the processes, in turn.

A. Back-end Process

Step 1: Pick the next element from the back-end queue. If the queue is empty, then go to step 12.

Step 2: Examine the type of the element. If element is of type 2, then go to step 3 and perform directory processing. If element is of type 5, then go to step 4 and accept corresponding descriptors. If element is of type 3, then go to step 10 and perform insert record.

Step 3: Delay for time to perform descriptor processing. Change the type of the element to 5 and insert it into the broadcast bus queue. Go to Step 11.

Step 4: Accept Corresponding Descriptors. Check if the corresponding descriptors for this request have been received from all the other (n-1) back-ends. If the request is an insert request, then delay for the time to perform the first step of address
generation. Generate an element of type 4 to represent a message from the back-end to the controller which indicates the cluster number of a record for insertion and go to step 11. If the request is a non-insert request, then delay for the time to perform the three steps of address generation. Access the tracks containing the records of the permitted set of clusters by placing I/O requests into disk drive queues. Wait until a disk drive indicates that a track of records has been accessed and placed in the main memory of the back-end. Then, process each such accessed record as in steps 5 through 9.

Step 5: Check each accessed record against the user query. Delay for the time to check a record against a single predicate multiplied by the number of predicates in the user query.

Step 6: If the request is a retrieve request, then go to step 7. If the request is an update request, go to Step 8. If the request is a delete request, go to step 9.

Step 7: If the record satisfies the user query, place it in a main memory buffer for output to the controller. Go to step 5 if there are more records to be retrieved. Else, return the records.
in the buffer to the controller using an element of type 6 and go to step 11.

**Step 8:** If the record satisfies the user query, update the record and place it in a main memory buffer for writing back to a disk drive. If the main memory buffer is full, initiate an I/O request to write the contents of the buffer to secondary store. Go to step 5 if there are more records to be updated. Else, send an update completed message using an element of type 6 to the controller and go to step 11.

**Step 9:** If the record satisfies the user query, mark the record as deleted and place it in a main memory buffer for writing back to a disk drive. If the main memory buffer is full, initiate an I/O request to write the contents of the buffer to secondary store. Go to step 5 if there are more records to be deleted. Else, send a delete completed message to the controller using an element of type 6 and go to step 11.

**Step 10:** Delay for the time to perform the second and third steps of address generation. Then, generate an I/O request to the disk drive selected for inserting the record. On receipt of message from disk drive indicating successful completion of
insertion, send an insert completed message to the controller using an element of type 6 and go to step 11.

Step 11: Go to Step 1.

Step 12: Wait for next element to arrive to the back-end queue. When element arrives, go to step 1.

B. Disk Drive Process

Step 1: Pick the next element from the disk drive queue. If queue is empty, go to step 6.

Step 2: Examine the track number and cylinder number to which head must be moved to complete the I/O request represented by the element. Delay by the time to move the head to the appropriate track.

Step 3: Wait until back-end unibus becomes free.

Step 4: Delay for track rotation time.

Step 5: Go to Step 1.

Step 6: Wait for next element to arrive to the disk drive queue. When element arrives, go to step 1.
Controller Process

Step 1: Pick the next element from the controller queue. If queue is empty, go to step 7.

Step 2: Examine the type of the element. If the element is of type 1, then go to step 3. If the element is of type 4, then go to step 4. If the element is of type 6, then go to step 5.

Step 3: Delay for the time to parse the request and to broadcast it to all the back-ends. Place a type 2 element in the Vax Unibus queue for eventual transmission to the n back-ends. Go to step 6.

Step 4: Check if \((n-1)\) such elements have already been processed. If not, then go to step 6. Else, delay for time to search CINBT. Generate a type 3 element to represent a message to be sent to the back-end chosen for record insertion. Place this element in the Vax Unibus queue. Go to step 6.

Step 5: Check if \((n-1)\) such elements have already been received. If not, then go to step 6. Else, generate a type 7 element to represent the response to be returned to the user and place it in the Vax Unibus queue. Go to step 6.

Step 6: Go to Step 1.

Step 7: Wait for next element to arrive to the controller.
queue. When it arrives, go to step 1.

D. Broadcast Bus Process

Step 1: Pick the next element from the broadcast bus queue. If queue is empty, go to step 8.

Step 2: If the type of the element is 4 or 6, go to step 3. If the type of the element is 5, go to step 4. If the type of the element is 3, go to step 5. Else, if the element is of type 2, go to step 6.


Step 4: Generate (n-1) elements of type 5 to be stored in the queues of the (n-1) back-ends other than the one which placed the element in the broadcast bus queue. Delay for bus transmission time. Go to step 7.

Step 5: Place the element in the queue associated with the back-end chosen for insertion. Delay for bus transmission time. Go to step 7.

Step 6: Generate n elements of type 2 for the queues of all the n back-ends. Delay for bus transmission time. Go to step 7.

Step 7: Go to step 1.
Step 8: Wait for the next element to arrive to the broadcast bus queue. When it arrives, go to step 1.

E. Vax Unibus Process

Step 1: Pick the next element from the Vax Unibus queue. If queue is empty, go to step 7.

Step 2: If the type of the element is 4, 6 or 1, then go to step 3. If the type of the element is 7, then go to step 4. Else, go to step 5.

Step 3: Delay for the Vax Unibus transmission time. Place the element into the controller queue. Go to step 6.

Step 4: delay for the vax Unibus transmission time. Go to step 6.

Step 5: Delay for the Vax Unibus transmission time. Place the element into the broadcast bus queue. Go to step 6.

Step 6: Go to step 1.

Step 7: Wait for next element to arrive to the Vax Unibus queue. When it arrives, go to step 1.
APPENDIX I

INPUT PARAMETERS OF THE SIMULATION MODEL

The following are the input parameters of the simulation model used in Chapter 7. Each of these parameters has to be specified for every simulation run.

A. Parameters Related to Requests

Percentage of retrieve requests
Percentage of update requests
Percentage of delete requests
Percentage of insert requests
Interarrival time of requests
Number of predicates in the query of a request
Number of clusters to be processed for a request
B. Parameters Related to the Disk Drives

- Seek time
- Rotation time
- Number of disk drives per back-end
- Number of cylinders per disk drive
- Number of tracks per cylinder
- Number of bytes per track

C. Parameters Related to the Database

- Number of clusters
- Average cluster size

D. Parameters Related to the System

- Number of back-ends
- Broadcast bus speed
- Vax Unibus speed
- Time to send a message
- Time to parse a request
- Time to check if a record satisfies a predicate