INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify 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 complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of “sectioning” the material has been followed. 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 illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.
Orooji, Ali

EXERCISES AND EXAMINATIONS OF SOFTWARE ENGINEERING TECHNIQUES FOR THE IMPLEMENTATION OF LARGE-SCALE DATABASE SYSTEMS: THE RESULTS OF A MULTI-BACKEND DATABASE SYSTEM IMPLEMENTATION

The Ohio State University

University Microfilms International
300 N. Zeeb Road, Ann Arbor, MI 48106
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 ___________________________________________
EXERCISES AND EXAMINATIONS OF SOFTWARE ENGINEERING TECHNIQUES
FOR THE IMPLEMENTATION OF LARGE-SCALE DATABASE SYSTEMS:
THE RESULTS OF A MULTI-BACKEND DATABASE SYSTEM IMPLEMENTATION
DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

By
Ali Orooji, B.S., M.S.

* * * * *

The Ohio State University
1984

Reading Committee:
Dr. Douglas S. Kerr
Dr. David K. Hsiao
Dr. Jay Ramanathan

Approved By

Advisor
Department of Computer and Information Science
To my parents
I am very grateful to my advisor, Dr. Douglas S. Kerr, for his enormous support, guidance, patience, concern, understanding and help throughout this research. I would also like to thank him for greatly improving my writing ability. I can never thank him enough for the time he devoted to this research in the last three years. I would like to thank Dr. David K. Hsiao for treating me like one of his advisees and for spending so much time in helping me. I would also like to thank Dr. Jay Ramanathan for reading this report and for her useful suggestions.

I would like to acknowledge that this work was supported by Contract N00014-84-WR-24058 from the Office of Naval Research to Dr. David K. Hsiao and conducted in the Laboratory for Database Systems Research. The Laboratory for Database Systems Research is initially funded by the Digital Equipment Corporation (DEC), Office of Naval Research (ONR) and the Ohio State University (OSU).
laboratory consists of staff, graduate students, undergraduate students, visiting scholars and faculties for conducting research in database systems. The Laboratory for Database Systems Research was moved from OSU to the Naval Postgraduate School (NPS) in June of 1983, and it is currently supported by ONR and NPS.

I would like to thank all those who have helped with the multi-backend database system (MDBS) project. In particular, the MDBS design and analysis were developed by Jai Menon. (Now, Dr. Jai Menon of IBM Research Laboratory, San Jose, California.) Several graduate students, visiting scholars and undergraduate students provided much help in the detailed design and coding: Steven Barth, Julie Bendig, Abdulrahim Beram, Richard Boyne, Raymond Browder, Steven Demurjian, Patti Dock, Xing-Gui He, Masanobu Higashida, Chris Jeschke, Jim Kiper, Drew Logan, Jim McKenna, William Mielke, Tamer Ozsu, Zong-Zhi Shi, Paula Strawser, and Joe Stuber. Jose Alegria, Tom Bodnovich and David Brown contributed background material which was necessary for making our design decisions. I would also like to thank the laboratory staff and other operators who provided us with system support: Bill Donovan, Doug Karl, Paul Placeway, Steve Romig, Jim Skon, Dennis Slaggy, Mark Verber, and Geoff Wyant at the Ohio State University; and Al Wong and his
staff at the Naval Postgraduate School.

Last but not the least, I would like to thank the Ohio State University and the Naval Postgraduate School for providing ideal environments for me to carry out this research.
VITA

October 14, 1953 .... Born - Tehran, Iran

June, 1976 ........ B.S. Mathematics, College of Mathematics and Economic Management, Iran

December, 1978 .... M.S. Computer Science, Florida Institute of Technology, Melbourne, Florida, U.S.A.

1980-1984 .......... Graduate Assistant, Department of Computer and Information Science, The Ohio State University, Columbus, Ohio, U.S.A.

PUBLICATIONS


FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in Operating and Database Systems. Professor Douglas S. Kerr and Professor David K. Hsiao

Studies in Programming Languages. Professor Jay Ramanathan

Studies in Computer Architecture. Professor Ming T. Liu
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEDICATION</td>
<td>ii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENT</td>
<td>iii</td>
</tr>
<tr>
<td>VITA</td>
<td>vi</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>viii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xiv</td>
</tr>
<tr>
<td>Chapter</td>
<td></td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 The Profile and Goals of the Experiment</td>
<td>2</td>
</tr>
<tr>
<td>1.2 An Effort in Engineering Database Systems</td>
<td>3</td>
</tr>
<tr>
<td>1.3 The Organization of the Remaining Chapters</td>
<td>6</td>
</tr>
<tr>
<td>2. IMPLEMENTATION CONSIDERATIONS OF LARGE-SCALE DATABASE SYSTEMS</td>
<td>8</td>
</tr>
<tr>
<td>2.1 The Management Issues</td>
<td>10</td>
</tr>
<tr>
<td>2.1.1 The Choice of an Implementation Strategy</td>
<td>12</td>
</tr>
<tr>
<td>2.1.2 The Organization of Implementation Personnel</td>
<td>13</td>
</tr>
<tr>
<td>2.1.3 The Training and Expertise of Implementors</td>
<td>14</td>
</tr>
<tr>
<td>2.1.4 The Turnover of Programming Teams and Implementation Environment</td>
<td>15</td>
</tr>
<tr>
<td>2.1.5 The Assurance of Software Quality</td>
<td>17</td>
</tr>
</tbody>
</table>

viii
2.2 The Issues of System Hardware and Software .... 19
  2.2.1 The Selection of Hardware ............... 22
  2.2.2 The Choice of System Implementation
    Language and Operating System Interface ... 23
  2.2.3 The Choice of Operating Systems and the
    Impact of Software Tools ................ 25
  2.2.4 The Techniques for Software Development .... 27
2.3 The Issues of Database System Implementation .. 28
  2.3.1 The Requirements of Hardware and
    Software ................................ 29
  2.3.2 The Restrictions Imposed by the
    Hardware ................................ 32
  2.3.3 The Interface with the Operating
    Systems .................................. 33
  2.3.4 The Alternative Levels of Input/Output
    Operations .............................. 34
  2.3.5 The Need for Message-Passing Mechanisms ... 34
  2.3.6 The Organization of Directory Data and
    User Data ................................ 35
  2.3.7 The Concept and Implementation of
    Concurrency Control ........................ 37
  2.3.8 The Need for and Impact of Additional
    Software .................................. 38
3. SOLUTIONS TO THE MANAGEMENT ISSUES ........ 39
  3.1 The Implementation Strategy .................. 40
    3.1.1 Typical Strategies: Build-it-twice Full-
      prototype, Level-by-level Top-down,
      Incremental Development, and
      Advancement ................................ 41
    3.1.2 The DBSIM Strategy: Application of
      Scaffolding, Incremental Development and
      Build-some-twice-different-versions
      Approaches ................................ 44
      3.1.2.1 The Original Implementation
        Strategy ................................ 50
      3.1.2.2 The Revised Implementation
        Strategy ................................ 52
    3.1.3 A Critique of the DBSIM Implementation
      Strategy ................................ 56
  3.2 The Team Organization ....................... 59
    3.2.1 Typical Organizations: Democratic
      Decentralized, Controlled Centralized,
      and Controlled Decentralized .......... 60
    3.2.2 The DBSIM Organization: Three-level
      Chief-programmer-team without
      Librarians ................................ 66
3.2.3 A Critique of the DBSIM Team
Organization ........................................ 68
3.3 The Software Quality Assurance (SQA) ............ 70
  3.3.1 Typical SQA Techniques: Design and Code
    Inspections, Reviews, Audits, and
    Structured Walkthroughs ........................ 71
  3.3.2 The SQA Techniques Used in DBSIM: Formal
    Reviews for Design and Code ....................... 72
  3.3.3 A Critique of the SQA Techniques Used in
    DBSIM ............................................ 73
3.4 The Need for Periodic Overall System Reviews ... 77
  3.4.1 The Problem with Concurrency Control .......... 77
  3.4.2 The Concept of Overall System Reviews ... 79
3.5 Concluding Remarks on the Management
Techniques ........................................... 81

4. SOLUTIONS TO THE ISSUES OF SYSTEM HARDWARE AND
SOFTWARE ............................................. 83
  4.1 The System Hardware: VAX-11/780, PDP-11/44,
and PCL ............................................. 83
  4.2 The Systems Programming Language ............... 86
    4.2.1 Typical Languages: Bliss, C, and
      MAINSAIL .................................. 86
      4.2.1.1 The Bliss Language and Its
        Compilers .................................. 88
      4.2.1.2 The C Language and Its
        Programming Development
        Environment ............................... 89
      4.2.1.3 The MAINSAIL Language and Its
        Relationship to the other
        Languages ................................ 90
    4.2.2 The Language Used in MDBS
      Implementation: C ........................... 92
    4.2.3 A Critique of the Language Used in MDBS
      Implementation .............................. 93
  4.3 The Development-Time and Run-Time Operating
Systems ............................................. 94
    4.3.1 The Operating Systems Used in MDBS
      Implementation: UNIX, RSX, and VMS .......... 95
    4.3.2 A Critique of the Operating Systems Used
      in MDBS Implementation ....................... 97
  4.4 Software Development Techniques .................. 101
  4.4.1 Design ..................................... 102
    4.4.1.1 Typical Techniques: Top-Down,
      Bottom-Up, and
      Data-Structure-Driven ....................... 102
4.4.1.2 The DBSIM Techniques .......... 104
   (A) Top-Down Design and Use of Data and Service Abstractions .......... 104
   (B) A Systems Specification Language (SSL) .......... 106
   (C) The Use of Jackson Charts ... 108
4.4.1.3 A Critique of the DBSIM Design Techniques .......... 114
4.4.2 Coding .................................... 116
   4.4.2.1 Typical Techniques: Principles of Modularity and Use of Structured Code .......... 116
   4.4.2.2 The DBSIM Techniques: Structured Coding Methodology, and Program Modification Control .......... 117
   4.4.2.3 A Critique of the DBSIM Coding Techniques .......... 118
4.4.3 Validation ................................ 120
   4.4.3.1 Typical Techniques: Verification and Testing .......... 120
   4.4.3.2 The DBSIM Technique: Black-box Testing .......... 122
   4.4.3.3 A Critique of the DBSIM Validation Technique .......... 123
4.5 Concluding Remarks on the System Hardware and Software .......... 125

5. SOLUTIONS TO THE ISSUES OF DATABASE SYSTEM IMPLEMENTATION .......... 128

5.1 Two Approaches to the Interface with the Operating System .......... 129
   5.1.1 The Choice of an Approach .......... 130
   5.1.2 The MDBS Process Structure .......... 131
5.2 The Inter-Process and Inter-Computer Message-Passing Mechanisms .......... 133
   5.2.1 Message Passing Within the Controller .......... 133
   5.2.2 Message Passing Within a Backend .......... 135
   5.2.3 Message Passing Between Computers .......... 136
5.3 Simulation of the Broadcasting Hardware .......... 138
5.4 The Input/Output Operations .......... 140
   5.4.1 The Size of Unit for I/O Operations .......... 141
   5.4.2 The Level of I/O Operations .......... 143
5.5 Implementations of Directory-Data and User-Data Structures .......... 145
   5.5.1 The Directory-Data Structures .......... 147
   5.5.2 The User-Data Structures .......... 148
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A Simplified Development Environment</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>The MDBS Hardware Organization</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>The MDBS Software Architecture</td>
<td>48</td>
</tr>
<tr>
<td>4</td>
<td>Original and Revised Implementation Plans</td>
<td>57</td>
</tr>
<tr>
<td>5</td>
<td>The Democratic Decentralized Team Organization</td>
<td>61</td>
</tr>
<tr>
<td>6</td>
<td>The Controlled Centralized Team Organization</td>
<td>63</td>
</tr>
<tr>
<td>7</td>
<td>The Controlled Decentralized Team Organization</td>
<td>65</td>
</tr>
<tr>
<td>8</td>
<td>The Organization of the MDBS Design and Implementation Teams as of 1/11/83</td>
<td>67</td>
</tr>
<tr>
<td>9</td>
<td>A Sample Walkthrough Report</td>
<td>74</td>
</tr>
<tr>
<td>10</td>
<td>An SSL Specification of a Program Procedure</td>
<td>109</td>
</tr>
<tr>
<td>11</td>
<td>The Constructs Used in a Jackson Chart</td>
<td>111</td>
</tr>
<tr>
<td>12</td>
<td>A Sample Program Structure</td>
<td>112</td>
</tr>
<tr>
<td>13</td>
<td>The SSL Corresponding to the Sample Program Structure in Figure 12</td>
<td>113</td>
</tr>
</tbody>
</table>
Figure 14: MDBS Versions and Completion Dates ........ 169
Figure 15: MDBS Development Chronology ............... 175
1. INTRODUCTION

Much of the research in software engineering has been directed at defining techniques and methodologies which contribute to the timely development of reliable software. Several techniques and methodologies for project management and for different phases of the software life cycle (requirements specification, design, coding and validation) have been proposed. These techniques and methodologies are intended to overcome the software crisis (cost, schedule and quality) [Boeh73].

The current techniques, tools and methodologies do indeed contribute to the timely development of reliable software. There are, however, few documented cases which provide facts and figures as tangible evidence of a software engineering success. More interestingly, the documented cases are instances of application software development, not system software development.

We have experimented with the application of software engineering techniques to the development of large-scale
database systems. In particular, a multi-backend database system, known as MDBS, has been implemented. The results of this experiment in database systems engineering are presented in this report.

1.1. The Profile and Goals of the Experiment

Although the concept and design of the multi-backend database system (MDBS) has begun in December of 1979, for its implementation a group is formed in June of 1981 at the Laboratory for Database Systems Research at The Ohio State University. One goal of the group is to investigate the role of software engineering techniques in the implementation of large-scale database systems. The plan is to first develop a database systems implementation methodology (DBSIM). The methodology is then to be used in the implementation of a prototype of the multi-backend database system (MDBS). The implementation of the system for experimental use is another goal of the group. The multi-backend database system (MDBS) is designed to allow performance improvement and capacity growth [Hsia81a, Hsia81b]. The analytical studies (modeling techniques for performance evaluation) in [Hsia81a, Hsia81b] show that MDBS is promising for supporting large databases. Our second
goal is to build a prototype of MDBS and carry out actual evaluation (measurement techniques for performance evaluation) of the system.

Now, in August, 1984, the experiment is complete; a prototype of MDBS has been developed using the database systems implementation methodology (DBSIM). We have had success with the techniques in DBSIM; we have also seen areas for improvement. In this report, we describe our database systems engineering experience. In particular, we describe the methodology used, effectiveness of the techniques in the methodology, and our recommendations for a more effective future implementation effort.

1.2. An Effort in Engineering Database Systems Software

In implementing the prototype MDBS, we first set out to develop a methodology for engineering database systems. This is interesting to us from a research standpoint for several reasons. First, can we develop a workable, comprehensive methodology? Second, can we make a qualitative contribution to database systems engineering by applying and evaluating our methodology throughout the implementation? Third, can we make a quantitative
contribution to database systems engineering by recording the facts and figures of our failures and successes.

We also have a practical motivation for conducting this exercise. The implementation teams are composed of graduate and undergraduate students, working part-time. The composition of the teams shift as students graduate or move on to other jobs. Team members require time to learn the methodology, and to acquire the requisite programming skills. The implementation group as a whole lack systems programming experience. The development environment is, for the first months of the effort, less than desirable owing to equipment delivery schedules. With these factors in mind, we have decided that an effort without structure is doomed to failure.

In order to make the best use of our resources, we have chosen to base our methodology on proven software engineering techniques. First and foremost, our goal is the development of reliable software according to specification and on schedule. But other considerations have entered into our choice of techniques. The techniques must be simple enough so that new team members can be integrated into the group with the least amount of effort. The effort must be
managed in such a way that the departure of a team member would not severely deplete group knowledge. Design techniques must be emphasized in an effort to minimize the coding task.

We have developed a database systems implementation methodology (DBSIM) with all these factors in mind. This methodology includes software engineering techniques for managing the database systems effort and for developing the database systems. The methodology can be summarized as follows:

Management Techniques
  Incremental Development
  Chief-Programmer-Team Organization
  Formal Reviews for Design and Code

Software Development Techniques
  Design Techniques
    Top-down Design and Use of Data and Service Abstractions
    Systems Specification Language
    Use of Jackson Chart
  Coding Techniques
    Principles of Structured Coding
    Program Modification Control
  Validation Techniques
    Black-box Testing Approach

Since we have begun the MDBS development using the requirements specification presented in [Hsia81a, Hsia81b], we have not considered techniques for the requirements specification phase of the software life cycle.
In this report, we describe our database systems engineering experience. In particular, we describe the methodology used, the applicability of software engineering techniques to the development of database systems, the problems we observed, and our recommendations for a more effective future implementation effort. We will not present the design, analysis and implementation details of the database system, MDBS. Details on the design, analysis and implementation of MDBS have been reported in [Hsia81a, Hsia81b, He83, Kerr83, Boyn83a, Boyn83b, Demu84].

1.3. The Organization of the Remaining Chapters

We describe the database systems implementation methodology (DBSIM) and our experience in implementing the multi-backend database system (MDBS) in the rest of this report. In Chapter 2, we introduce different issues that must be addressed for a database system development effort. We categorize these issues as "management issues", "system hardware and software issues" and "database system implementation issues."

A review of different solutions to the three categories of issues and our solutions to these issues are described in
Chapters 3, 4 and 5, respectively. Our database systems engineering experience is also described in detail in these chapters. In Chapter 6, we summarize the database systems implementation methodology (DBSIM) and our experience. Finally, in Chapter 7, we present our conclusions.
2. IMPLEMENTATION CONSIDERATIONS OF LARGE-SCALE DATABASE SYSTEMS

Several issues must be addressed for a database system development effort. The project planners need to decide how the system is going to be built. Is it going to be built in stages or is the complete system going to be built at once? Development time and chance of success may vary for different implementation strategies. The expertise of the implementors can also affect the decision on an implementation strategy. The project planners need to choose a team organization; will all implementors be in one team or will there be several teams? How will the implementors be supervised? The project planners also need to adopt procedures to assure that the final product meets certain requirements. We categorize these and similar issues as "management issues."

Since a database system is going to be implemented, project planners need to decide on the hardware for the system. Hardware cost and system performance requirements
must be considered in making this selection. The project planners also need to choose techniques for different phases of the software life cycle. What are the design techniques? What are the constructs that should be used in coding? How is the system going to be tested? These techniques should result in the timely development of reliable software. We categorize these and similar issues as "system hardware and software issues."

The management and system hardware and software issues apply to all database system development efforts. There are, however, some issues that are specific to each system and vary from database system to database system. For example, the implementors need to decide how directory data will be represented. In one database system, the implementors may decide to store the directory data the same way as the user data; in another database system, the implementors may decide, for performance reasons, to use a different organization for the directory data. What type of I/O operations will be used? What operating system features will be used? These choices affect the performance of the database system. We categorize these and similar issues as "database system implementation issues."
In this chapter, we describe the issues that must be addressed for a database system development effort. The solutions to these issues are discussed in Chapters 3, 4 and 5.

2.1. The Management Issues

Management of a task, such as an implementation effort, is an important key to the success or failure of the task. As an example, let us consider the management issue of assigning jobs to personnel. With proper job assignment, maximum use of available manpower will be made and the task will be completed successfully. With a poor job assignment, on the other hand, the task will most likely not complete in time and when it is completed, if it is ever completed, the final product will not be nearly as satisfactory as intended. Job assignment is related to team organization. Thus, the project planners need to decide how to organize the personnel; all the implementors can be grouped into one team or they can be divided into several teams. If there will be several teams, the level of interactions between teams should be determined.
A task can not be accomplished without a plan. Thus, the first decision the project planners are faced with is the choice of an implementation strategy, i.e., how the final system will be developed. The final system can be developed in stages or it can be developed at once. The expertise of the team members can affect this choice. For example, an inexperienced team has more chance of success in accomplishing a simple task than in accomplishing a complicated task. Thus, if the team members are inexperienced and the system is large, it may be advisable to develop the system in stages.

The project planners also need to adopt some procedures to assure that the final product meets certain requirements. If the final product does not satisfy its requirements, it is, obviously, not acceptable. Thus, the implementation effort would be useless and a failure. In order to avoid such a disaster, project planners need to adopt procedures to control the quality of the product being built at each stage of its development.

In the following sections, we discuss these management issues in more detail. The solutions to these issues are discussed in Chapter 3.
2.1.1. The Choice of an Implementation Strategy

Many different implementation strategies have been proposed. The incremental development approach and the level-by-level top-down approach are among the widely used implementation strategies. The strategy used in implementing a system can be:

(1) One of the already-proposed strategies,
(2) A combination of the old strategies,
(3) One or a combination of the old strategies with modifications,
(4) A new strategy, or
(5) A combination of the old and new strategies.

The advantage of adopting the implementation strategy 1, 2 or 3 is that, since it is an old strategy, it has most likely been used before in several implementation efforts and its effectiveness has been reported. Thus, an appropriate implementation strategy can be selected. The strategy 4 or 5 is, however, needed if none of the old strategies is appropriate for an implementation effort.

Several factors should be considered when selecting an implementation strategy. The expertise of the team members is the primary factor. If the team members are inexperienced, their first few months should be spent in
learning the system. They can also implement some of the
supporting software, e.g., a program for generating test
files. After gaining experience, they can participate in
implementing the database system. Time and cost constraints
further complicate the choice of an implementation strategy.
Some systems are not useful if they are not completed in
time, e.g., if some supporting software for the current
hardware technology is completed after the next generation
of hardware technology is introduced and the current
technology is obsolete, the supporting software is
practically useless. Cost is also a constraint since an
implementation effort may have to be stopped if it exceeds
its budget.

The project planners need to select an appropriate
implementation strategy. In summary, the strategy chosen
should assure timely and cost effective development of the
system in consideration of the experience or inexperience of
the implementation team members.

2.1.2. The Organization of Implementation Personnel

Several people participate in implementing a large-
scale system. Thus, a team organization is required. The
organization should allow the maximum use of the available
manpower. At the same time, it should guarantee the development of a well-designed and well-structured system (the characteristics of a well-designed and well-structured system are described in Section 2.1.5).

The implementation team members are to be affected by the choice of an organization, and vice versa. If the team members are continuously changing, the organization should allow the easy integration of the new members. It should also have the provision that the departure of a team member does not result in the loss of (or a break in) his or her subproject.

The project planners need to select a team organization. The primary concern in choosing an organization, as in choosing an implementation strategy, is to assure that the system will be developed in time and cost effectively. The expertise of the team members and the frequency at which team members change can affect the choice of a team organization.

2.1.3. The Training and Expertise of Implementors

As described in the previous two sections, the implementation team members affect every aspect of a
development effort. If the team members are inexperienced, they need to be trained before they can participate in implementing the system. If the team members have experience, their expertise should be used towards a more timely implementation of the system.

It is interesting to note that companies consider a new employee's first 6-12 months as an investment, i.e., they pay full salary to the new employee just to get trained. It is after this period that a new employee can do productive work for a company. Thus, it is a reasonable estimate to say that it takes 6-18 months longer to implement a large-scale system with inexperienced team members than with experienced members. The variation (6-18 months) depends on the frequency at which team members leave the project and new members join the project. It is desirable to keep the effects of a change in project members to a minimum, i.e., provision should be provided to minimize the effects of the departure of a team member.

2.1.4. The Turnover of Programming Teams and Implementation Environment

As pointed out in the previous sections, the time that it takes to implement a large-scale system varies depending
on the expertise of the team members and the frequency at which team members change. The team members are from the implementation environment (university or company). The multi-backend database system (MDBS) has been implemented in a university. So, we discuss only the problems in a university environment.

The MDBS project members are faculty and students. The project members have other responsibilities, such as teaching and/or taking courses. Thus, they are not working solely on the project.

Initially, we had no members with experience in systems programming and knowledge of operating systems. The training has taken a major portion of the implementation time. The team members are also constantly changing as the old members graduate and leave the project.

More than 20 people have participated in the MDBS development. The majority of the undergraduate students have stayed on the project less than a year (they have either graduated or wished to leave the project). Considering that a new member's first 3-6 months is spent in training and learning the system, the average turnover for the project is estimated at 50%. Thus, the turnover rate is
very high at a university environment, and provision is needed to allow continuous progress in an implementation effort carried out in a university.

2.1.5. The Assurance of Software Quality

The goal of an implementation effort is to develop a system that satisfies its requirements. A system is not, of course, acceptable if it does not meet its requirements. For example, let us say that a system is required to support multiple users. This system is not, obviously, acceptable if it can support only a single user. As another example, a real-time system, such as one that monitors the transfusion of blood into a patient's body, is required to respond immediately to a request (changes in environment). A real-time system is not acceptable if it does not respond in a matter of milliseconds (or even nanoseconds depending on what the system does); what good it would do to a patient to receive blood, say, two hours after he needed it! As an example in database system field, all database systems are required to handle insertion and retrieval of records. A database system is not, obviously, acceptable if it can not support insertion and/or retrieval of records.
It is also necessary that the end product, for all systems in general and for large-scale systems in particular, have a high quality, i.e., it should be well designed and well structured. A high-quality, well-designed and well-structured software is one that is simple, reliable, understandable and maintainable. These characteristics are defined as follows [Jens79]:

- Simplicity: The product is easy to read and comprehend.

- Reliability: The product will perform the intended user's functions accurately under normal conditions and will do so consistently and completely. Abnormal conditions may cause degraded performance, but will not result in erroneous performance masked as correct performance.

- Understandability: The product package is understandable to the extent that the user can easily grasp the functioning of the product and the relationship between the product and other products and system components. There should be no hidden meanings or operating characteristics that come to light only after months or years of use.

- Maintainability: A product is maintainable when it is easily understood by the maintenance programmers and is easy to modify and test when updating to meet new requirements, rectifying a deficiency, correcting errors, or moving to a different but similar computer system.

Several people participate in implementing a large-scale system. Techniques for assuring the quality of the software produced by team members are required. The
techniques must be employed throughout the implementation effort in order to produce high-quality software. Team members must be supervised to assure that they employ the techniques and that the software produced by them is well designed and well structured. This is not, obviously, an easy task.

2.2. The Issues of System Hardware and Software

A simplified development environment is depicted in Figure 1. Using this environment, application programs are written in assembly or a high-level language. Thus, an application programmer, a developer, is only concerned with a programming language and he is not concerned with the rest of the system shown in Figure 1.

When developing a database system, the situation is not as simple. The developer may have to be concerned with all the components shown in Figure 1, i.e., he may have to choose the system hardware, the underlying operating system, and the systems programming language. A special hardware configuration, as opposed to the typical general-purpose computer systems, may be needed for the database system being developed, so the project planners need to choose the
Figure 1: A Simplified Development Environment
system hardware. For example, the database system may require special-purpose hardware that is intended to perform search and sort operations. The database system also requires the underlying operating system to have some features, so the underlying operating system must be selected. For example, the database system may support multiple users and concurrent request execution. Thus, an operating system which provides mechanisms for supporting multiple processes must be chosen. The project planners also need to select a systems programming language for coding the system.

Effective software development techniques are needed for a large-scale implementation effort to be successful. The techniques are needed to avoid software crisis (cost, schedule and quality) [Boeh73]. Thus, techniques for different phases of the software life cycle (requirements specification, design, coding and validation) must also be selected for a database system implementation effort.

In the following sections, we discuss these system hardware and software issues in more detail. The solutions to these issues are discussed in Chapter 4.
2.2.1. The Selection of Hardware

Many recently proposed large-scale database systems are designed to allow performance improvement and capacity growth. These database systems use multiple computers and multiple disk drives. The performance improvement is achieved by increasing number of computers in the system. The capacity growth is achieved by increasing number of disk drives. The typical general-purpose computer systems may not be satisfactory for a database system. For example, the database system may require a hardware which performs the search operation very efficiently. A special-purpose hardware would then be more suitable than a general-purpose hardware. Thus, the project planners need to choose hardware for different system components.

If there are multiple computers in the database system, they must communicate with each other. The project planners must consider communication hardware. In this case, hardware for connecting the computers together may also have to be chosen.

Two factors affect the choice of system hardware: cost and performance requirements. For the system cost, the database system developer must consider the original cost
and the system expansion cost (multiple-computer database systems are generally configured in such a way that more computers and disk drives can be integrated into the system).

Performance requirements of a database system may restrict the hardware choices. The system response time and its storage capacity are the primary requirements.

2.2.2. The Choice of System Implementation Language and Operating System Interface

Referring to Figure 1, a language or languages must be chosen for implementing the database system. To make programming easier, a single high-level language is usually selected for this purpose. The developer may, however, have to write some parts in assembly language. This happens if the language chosen does not have enough constructs to code the system. For example, the database system may require an interface to the disk driver. If the systems programming language chosen does not have constructs for coding the interface, the developer has to write it in assembly language.
The factors that affect the choice of a systems programming language include: the efficiency, simplicity, power and portability of the language, and the programming environment for the language. The language chosen must have enough constructs to program the features of the database system being implemented. At the same time, it should be easy to use and easy to understand.

The efficiency of a language is determined by the time required to run a program written in that language, as compared to the time required to run the same program written in another language. The portability of a language is determined by how system dependent the programs written in that language are, i.e., the portability is determined by the amount of changes to the programs that is required when transferring the programs from one system to another. A portable language will require very few changes to the programs, i.e., the programs written in that language are not system dependent. The efficiency of the systems programming language has direct effects on the overall performance of the database system. Thus, an efficient language is desirable. It is also desirable for the database system software to be portable. Thus, the language portability is an important factor.
A programming environment is a set of integrated tools that helps the programmers to develop their programs. The programming environment provided to a user determines the programming effort needed to accomplish a task. A good programming environment greatly reduces the programmer's effort and it is, therefore, desirable.

A database system needs operating system-related operations such as I/O, process synchronization and process communication. A database system implementor also has to decide what facilities provided by the underlying operating system to use, and what facilities to implement himself. To minimize the programming effort, the operating system-provided facilities, as much as possible, should be used. Some of the facilities provided by the operating system, such as the I/O operations, may not, however, be satisfactory, for performance reasons. So, the implementor has to write his own.

2.2.3. The Choice of Operating Systems and The Impact of Software Tools

The multiple computers in a large-scale database system may run on the same operating system or they may run on different operating systems. Thus, the database system
implementor has to choose one or more run-time operating systems. The primary factor in the selection is the efficiency of the operating systems since the performance of the database system depends also on the underlying operating systems. The database system also requires the operating systems to have some features, e.g., the capability for supporting multiple processes. These requirements also affect the selection.

Another decision to be made by the database system implementor is that of development-time operating system. This operating system need not be the same as the run-time operating systems. The development-time operating system is used by the implementor during the design and coding of the database system. The run-time operating systems are used by the database system at operation time. A user-friendly operating system can be used during the development provided that the software produced on the development-time operating system is portable to the run-time operating systems. Availability of software tools to help the implementor develop his system is the main characteristic of a user-friendly development-time operating system.
In summary, the run-time operating system must be an efficient one which can support the new database system effectively and speedily. The development-time operating system, on the other hand, must be a user-friendly one which can support the database implementor and reduce his effort.

2.2.4. The Techniques for Software Development

The four phases of the software development life cycle are defined as follows [Leat83]:

(1) Requirements Specification - This phase involves stating the purpose of the software: what is to be done, not how it is to be done.

(2) Design - During this phase an algorithm (method) is devised to carry out the specification produced in the previous phase. That is, how to implement the system is specified during this phase.

(3) Coding - During this phase the design is translated into a programming language.

(4) Validation - During this phase it is ensured that the developed system functions as originally intended. That is, it is validated that the system actually performs what it is supposed to do.

Effective software engineering techniques must be selected for these stages. The main objective of the techniques should be to enhance the quality of the software which is developed. A further objective is that the implementation should proceed as quickly and effectively as possible. The
database system implementors must employ the selected software engineering techniques, and the effectiveness of the techniques must be monitored. Other software development techniques may be added during the implementation effort to complement the original techniques.

2.3. The Issues of Database System Implementation

The issues addressed in Sections 2.1 and 2.2 apply to all large-scale database systems. There are, however, some issues that may vary from database system to database system. For example, the hardware and software requirements may vary from a database system to another database system. One multi-computer database system may require identical hardware and replicated software, while another may require a network of dissimilar computers running their own specific software. Another example would be the organization of directory data. As described before, different database systems may have different directory-data organizations.

In the following sections, we discuss the database system implementation issues in more detail. The solutions to these issues are discussed in Chapter 5. The issues addressed in the following sections are specific to the
database system that we have implemented, namely, MDBS. Some understanding of the multi-backend database system (MDBS) is necessary to understand the issues discussed here (and the software engineering issues discussed later). Thus, we first discuss the hardware and software requirements, i.e., the hardware and software characteristics of MDBS.

2.3.1. The Requirements of Hardware and Software

The purpose of this report is to describe the database systems engineering experience rather than the particular database system being implemented. So, the description presented here is brief and informal. Full details on the design and analysis of MDBS can be found in [Hsia81a, Hsia81b]. The implementation effort is described in more detail in [He83, Kerr83, Boyn83a, Boyn83b, Demu84].

The MDBS hardware organization is shown in Figure 2. MDBS is connected to a host computer through the controller. The controller and backends are connected by a broadcast bus. The controller receives requests from a host computer, and broadcasts each request to all backends. The database is distributed across all the backends in such a way that, on the average, equal portions of the relevant data for a
Figure 2: The MDBS Hardware Organization
request reside at each backend. Scheduling and concurrency control are implemented at the backends. Although the backends operate separately with some coordination, they do not operate in a lock-step mode.

MDBS is designed to allow performance improvement and capacity growth. Performance improvement is achieved by adding new backends to the system. The backends run identical software, so the addition of a new backend does not require any new programming; the existing software is simply replicated on the new backend. The second design goal of MDBS, allowing capacity growth, is achieved by adding new disk drives to the system.

A major design goal of MDBS is to minimize the task of the controller. Thus, most of the work has been delegated to the backends. The controller performs only those functions which are best performed centrally. For example, when processing an insert request, the controller selects a backend for storing the record being inserted (since the data is distributed, the insert only occurs at one of the backends). The backends perform the majority of the work required for processing a request. For example, they determine the records relevant to a request, retrieve the
records, and extract required attribute values from the retrieved records.

2.3.2. The Restrictions Imposed by the Hardware

The hardware cost of a backend in MDBS determines the system expansion cost since MDBS is expanded by adding new backends to the system. Thus, low cost hardware must be chosen for backends to make system expansion feasible. The MDBS design described in [Hsia81a, Hsia81b] suggests the use of minicomputers which drive the hard disks as backends. The small physical memory size and process address space of typical minicomputers necessitates the use of overlays in the MDBS backends. The use of overlays affects the system performance. It also mandates special considerations in the implementation of the processes in the backends.

The MDBS design also requires a broadcasting bus, for connecting the controller and the backends. Since a broadcasting bus is not originally available, we must find an appropriate replacement. In choosing the replacement hardware, we must consider the MDBS design criteria that the addition of new backends will not increase communication overhead in the system.
2.3.3. The Interface with the Operating Systems

After the operating systems for the MDBS controller and backends have been selected, an interface with them must be chosen for supporting concurrent execution of multiple requests.

Operating systems have been characterized as either message-oriented or procedure-oriented, depending on how they implement the notions of process and synchronization [Laue79]. One of these approaches should be used in MDBS.

Using a message-oriented operating system, there would be a fixed number of processes (one per MDBS function). Synchronization is implemented by passing messages among processes. There is a relatively limited amount of direct sharing of data in the memory among processes. Processes for each activity are created when MDBS is started up. They are only deleted when MDBS is shut down.

Using a procedure-oriented operating system, there would be a varying number of processes (one per MDBS user). Synchronization is implemented by direct sharing and locking of common data in the main memory. Processes are rapidly created and deleted.
One of these approaches is used in a database system implementation. The facilities provided by the underlying operating system for each approach and ease of understanding of the database system must be considered in this selection.

2.3.4. The Alternative Levels of Input/Output Operations

Different levels of I/O operations are usually provided by operating systems. At the highest level of abstraction is the concept of files, where the user issues a read (write) of a record from (to) a file. The user is not concerned about physical addresses for the records on the disk. At the lowest level of abstraction, the user implements his own disk driver. He has to provide and keep track of physical addresses on the disk. There are other levels of abstraction between these two extreme levels as well. For MDBS, an I/O level must be chosen. The goal is to minimize our work and, at the same time, meet the system requirements where, for example, similar records in the database be stored in close proximity.

2.3.5. The Need for Message-Passing Mechanisms

As described in Section 2.3.1, the MDBS design requires communications between computers (the controller and
backends). Furthermore, if a message-oriented approach to operating system is selected for MDBS, inter-process communication is also needed (see Section 2.3.3). Thus, mechanisms for inter-computer and possibly inter-process communication have to be designed and implemented. The facilities provided by the underlying operating systems can be used to reduce implementors' effort. Once again, our goal is to minimize our work and, at the same time, meet the performance requirements.

As described in Section 2.3.2, an alternative hardware must be used in MDBS for connecting the computers since the broadcasting bus is not originally available. The new mechanisms for inter-computer communication must be carefully designed to minimize communication overhead.

2.3.6. The Organization of Directory Data and User Data

There are two types of data in most database systems: directory and user. The directory data is used to expedite access to the user data. Different organizations to represent and store the two types of data must be studied, and the appropriate organizations must be selected.
MDBS is based on the concept of clustering. Intuitively, a cluster is a set of similar records. For example, all the records for employees in the toy department and with salary between $10,000 and $20,000 may be one cluster. The directory data contains the cluster definitions and the secondary storage addresses for the clusters. The user data (or the database) contains the records in all the clusters defined in the system.

The performance of the directory management is a major factor in the performance of the system. Thus, a comprehensive study of different data organizations should be carried out, and considerable time should be devoted to the design of the directory management.

More than one implementation of the directory data and the user data may be carried out. The reasons for this are:

(1) The first implementation can be a simpler one as an exercise to gain some experience before working on a more complicated and efficient implementation.

(2) The performance of alternative implementations can be evaluated to choose the best one.
2.3.7. The Concept and Implementation of Concurrency Control

Concurrency control refers to the mechanisms that allow concurrent and interleaved executions of user requests while preserving the integrity and consistency of the database. Designing the concurrency control mechanisms for a database system and proving that it is correct is one of the most, if not the most, challenging tasks in database system design.

The concurrency control mechanism for MDBS is described in [Hsia81a, Hsia81b]. The task of the MDBS implementation group is to design and implement algorithms to carry out the concurrency control mechanism. Access to the directory data as well as access to the user data may have to be controlled in order to provide correct results (responses) to the database system users. If directory-data concurrency control is also needed, it can be part of the user-data concurrency control, or it may operate separately from the user-data concurrency control. The correctness of different algorithms must be shown, and the most suitable one must be selected.
2.3.8. The Need for and Impact of Additional Software

One of the goals of our implementation effort is to validate the simulation results on MDBS described in [Hsia81a, Hsia81b]. We intend to run the prototype MDBS with program-generated databases and with actual databases. Program-generated requests and actual requests will be used in these experiments. Thus, we need supporting software such as a test-file-generation program. The supporting software can be developed before, during or after the MDBS implementation effort.

The decisions made in the design and implementation of the supporting software can affect the way MDBS should operate, and vice versa. For example, let us assume that a program (supporting software) has been implemented that generates records containing complex numbers. Then, MDBS must be able to handle complex numbers in order to use this support software. The current and future effects of the supporting software on MDBS must be carefully studied so that the implementations of the supporting software and MDBS do not conflict with each other. Otherwise, major changes to the developed supporting software and/or MDBS will be necessary.
3. SOLUTIONS TO THE MANAGEMENT ISSUES

The management issues have been introduced in Section 2.1. In particular, an implementation strategy and a team organization must be selected for developing a system. Furthermore, techniques for assuring the quality of the software produced by the team members are necessary. The project planners need to select techniques that will result in the timely and cost-effective development of the system. The expertise of the team members and the frequency at which the team members change should be considered in these selections.

In this chapter, we describe the management techniques proposed by different software development methodologies, and the techniques used in the database systems implementation methodology (DBSIM). The results of our experience in using the techniques are also given.
3.1. The Implementation Strategy

An implementation strategy refers to the approach taken to reach the final version of the system. The project planners need to select an appropriate implementation strategy. They should consider the expertise of the team members in this selection. If the team members are inexperienced, it may be more practical to develop some simpler versions of the system before working on the final version, i.e., some functionality of the system should be implemented in the latter stages of the development effort. The project planners may also choose that the final version should be the only version developed by the implementors.

The implementation strategy chosen should assure that the system will be developed in time and cost effectively. In some cases, a software system is not useful if it is not on schedule. In other cases, the late delivery increases the software cost. Cost constraints further complicates the situation; an implementation effort may have to be stopped if it exceeds its budget.
3.1.1. Typical Strategies: Build-it-twice Full-prototype, Level-by-level Top-down, Incremental Development, and Advancemanship

Many different implementation strategies have been proposed. Several such strategies are suggested in [Boeh81]. For example, the build-it-twice full-prototype approach is recommended when the implementation team is in a new undertaking. In this approach, the system is implemented twice. That is, a complete system is implemented. After the first implementation is finished, the development effort starts from the beginning and builds the complete system again. In this case, the first implementation gives insight that may make the second implementation much more satisfactory. In other words, during the first implementation of the system, the implementors will gain more experience, comprehend the system better, and find out peculiarities of the system. This will result in a much more improved second implementation. The extra time and cost that may be needed for this strategy must be justified.

A frequently suggested approach is the level-by-level top-down approach. In this approach, the system being
developed is successively decomposed into smaller components (modules). At each level of decomposition, the modules of that level are developed and integrated. The next level of decomposition begins after the modules of the current level are developed and integrated. For example, the "request processing" module in a database system may be decomposed as follows:

```
+----------------+----+  + ------------+
  i  i i i   i  i  *  *
 insert retrieve delete update
 processing processing processing processing
```

The four modules at level 2 (insert processing, retrieve processing, delete processing and update processing) will be designed and integrated. Then, each of the four modules at level 2 can be decomposed, e.g., the "retrieve processing" module can be decomposed into "read tracks" and "extract values" modules. The decomposition of the level-2 modules generates the modules at level 3. Now, the modules at level 3 will be designed and integrated into the system. This process is repeated until the system is completely developed. The level-by-level top-down approach allows integration of modules to occur as they are developed.
Thus, the need for a massive system integration phase is reduced. The disadvantage of this approach is that it is possible to get multiple solutions for the same module, since a module can be decomposed into smaller modules in more than one way.

Two refinements of the build-it-twice full-prototype approach and the level-by-level top-down approach have also been suggested [Boeh81]. The incremental development approach consists of developing several systems, with each succeeding system having increased functionality. For example, the data types supported by a compiler can be implemented incrementally as follows:

Version 1: integer and character string
Version 2: real and above
Version 3: user-defined types and above two

The advancement approach consists of two components. The first is to develop some anticipating documentation, i.e., user documentation is developed before the system is actually developed. The second, called software scaffolding, consists of developing some of the supporting software before developing the actual system.
3.1.2. The DBSIM Strategy: Application of Scaffolding, Incremental Development and Build-some-twice-different-versions Approaches

We described in Section 2.1.1 that there are five cases for choosing an implementation strategy. One of these cases is "one or a combination of the old implementation strategies with modifications." Since we have decided to base our methodology on well-known software engineering techniques (see Chapter 1), the implementation strategy adopted for MDBS development is a combination of the old strategies with modifications. In particular, we have applied concepts of scaffolding since the implementors are inexperienced when they start. We have adopted the incremental development approach since it is easier to develop large systems in stages. Furthermore, this approach is useful when the implementation group is inexperienced, which is the case for our group. Finally, we have chosen to use the build-it-twice strategy with modification. More specifically, we have applied the concepts of this strategy in implementing only some components of the system and not the whole system, i.e., we have not build the complete system twice. Furthermore, as opposed to the build-it-twice strategy that suggests the two implementations be identical,
our two implementations of a component are not identical; our first implementation is a simpler version than the second implementation.

We have chosen to implement some of the supporting software as our initial effort. In particular, we have first implemented a program to generate test data files. We have also implemented a program to handle the initial loading of a database from an existing file. The early implementations of these less critical programs have allowed us to gain experience with our new computers, operating systems and programming languages before we have begun the implementation of the multi-backend database system itself. New team members have been integrated into the effort by assigning them work on less critical supporting software, building the software scaffolding. More specifically, we have a list of "useful and desirable" supporting software. Each new team member, especially a new programmer, is assigned to develop one of the supporting software as his first project.

We have chosen to implement five versions of MDBS, each with increasing functionality. Because we have been inexperienced when we begin, we have also adopted a build-
The build-it-twice approach in developing some components of MDBS. The first implementation would be a simplified version. This would give us experience to develop a more satisfactory second implementation. For example, the directory management has been built twice; a simple (but not acceptable in real applications) version that uses only the primary memory for storing the directory data, and a complicated (but required for real applications) version that uses the secondary memory for storing the directory data. The application of the build-it-twice strategy has also simplified the transition between different versions of MDBS. For example, we have implemented a simulated disk I/O and message-passing mechanism for the early versions of MDBS. The actual disk I/O and message-passing mechanism have been implemented in the subsequent versions of MDBS.

The original implementation strategy chosen is described in the following section. During the implementation we have modified this strategy somewhat. The new strategy and the reasons for modification are described in Section 3.1.2.2. Some understanding of the system under development, namely, MDBS, is necessary to understand the original and revised implementation strategies being presented here (and the software engineering issues
discussed later). Thus, we first give a functional description of MDBS.

The software architecture of MDBS is shown in Figure 3. Three categories of functions are performed by the controller. The request preparation functions receive, parse and format a request before broadcasting the formatted request to the backends. For consistency reasons, certain operations required for record insertion must also be performed in the controller. The insert information generation functions perform these operations. Since the data is distributed, the insert only occurs at one of the backends. The insert information generation functions select a backend for storing the record being inserted. The post processing functions collect the results for a request returned by each backend, and send them to the host machine.

A major design goal of MDBS is to minimize the task of the controller. Thus, most of the work has been delegated to the backends. There are three categories of functions performed by each backend (also in Figure 3). Since the backends run identical software, every backend performs the functions shown in Figure 3. The directory management functions determine the addresses of the records required to
Figure 3: The MDBS Software Architecture
process a particular request. The record processing functions perform record storage, record retrieval, record selection and attribute value extraction from the retrieved records. The concurrency control functions ensure that the concurrent and interleaved execution of user requests will keep the database consistent.

Let us now describe the boxes labeled communication interface in Figure 3. They represent the mechanism for communications between two functions in two different computers. There is a communication interface in each computer (the controller and the backends) since certain functions in each computer must communicate with certain functions in the other computers. The communication interface in each computer consists of two functions. The first function receives messages sent to its computer and forwards them to the specified MDBS functions within the computer. The second function receives messages (to be sent to other computers) from the MDBS functions in its computer and forwards them to the destination computers.

The original and revised implementation strategies described in Sections 3.1.2.1 and 3.1.2.2 refer to the hardware (VAX-11/780 and PDP-11/44s) and the operating
systems (VMS and RSX) used in the MDBS implementation. These are described in detail in Chapter 4. The strategies are also based on our implementation decision to use a separate process for each category of functions in MDBS. This implementation decision is also described in Chapter 4. Details on the hardware, operating systems, and MDBS process structure are not required to follow the material presented in this chapter.

3.1.2.1. The Original Implementation Strategy

The original implementation strategy has called for the implementation of five different versions of MDBS, beginning with a very simple system using a single minicomputer without concurrency control and with a simplified directory management, and ending with a full system including all the designed features.

As the first system, MDBS-I is to be a simple system running as a single process on a single computer. The purpose of this version is to allow the development of as much of the functionality of MDBS as possible in an environment that would allow easy development and debugging. It would allow us to delay the implementation of complex communication functions. Directory management is to be
simplified by assuming all directory information could be stored in the primary memory (note that the concept of the build-it-twice strategy is being used to simplify the first implementation of directory management).

The second version, MDBS-II, is to include concurrency control, but still be restricted to one minicomputer. The problem of inter-process communication is to be attacked. However we have planned to avoid the problems associated with inter-computer communication.

The third version, MDBS-III, is to run the same functions on multiple computers by adding the inter-computer communication functions. This is to be the first "real" system. The fourth version, MDBS-IV, is to include a "good" directory management system, one using secondary storage for the directory data (note that the concept of the build-it-twice strategy is being used to develop a more satisfactory second version of directory management). The fifth and final version, MDBS-V, is to include access control in the backends and a friendly user interface in the controller or in a host computer.
3.1.2.2. The Revised Implementation Strategy

As described in the previous section, the original implementation strategy has called for the implementation of five different versions of MDBS, beginning with a very simple system using a single minicomputer without concurrency control and with a simplified directory management, and ending with a full system including all the designed features. In the course of our implementation, we have chosen to revise our implementation strategy. We still want to implement Version III (the real system with simplified directory management) and Version IV (the real system with "good" directory management). However, we have chosen to modify the strategy of how to get to Version III. In order to avoid confusion, we will call the new versions A, B, etc.

Version A has been implemented on a VAX-11/780 running UNIX. It includes the request preparation and insert information generation functions of the controller. The post processing functions are not implemented. Instead, the output is displayed directly from record processing. The directory management and record processing functions of the backends are implemented, except that the aggregate
operations, such as average, minimum and maximum are omitted since they are straightforward but not critical to the overall system implementation effort. In addition, the routines that are to perform the actual input and output of data to and from the disks are not implemented. These routines are omitted because they are operating system dependent and this version is to run on a VAX using UNIX whereas the actual backends are to be PDP-11/44s using RSX-11M. Since the database is not to be stored on disks in this version, we have implemented a pseudo disk using the main memory (note that the concept of the build-it-twice strategy is being used to simplify the first implementation of the disk I/O). In addition a user test interface is implemented. This version which has been implemented as a single process will be called Version A.

The next step we have chosen, Version B, is to convert Version A to a multi-process, multi-computer system which has the same functionality as Version A. In this version, the controller has three processes: request preparation, insert information generation and post processing, and the backends have two processes: directory management and record processing. Concurrency control is to be added as a third process later. The conversion from a single process to a
multi-process implementation has required some modifications of the code in Version A. In particular, the programs that simulate message passing have been replaced by programs that send and receive messages between processes.

As the inter-process and inter-computer message passing facilities have become available, we have decided to modify our plans for the second version of MDBS. In particular, we have chosen to use two computers, the VAX and a PDP-11/44, rather than one. The final version of MDBS is to use PDP-11/44s (using RSX-11M) for the backends and a VAX (using VMS) for the controller. The inter-process communication facilities of VMS and RSX-11M are different. Thus implementing all the processes on one computer, the VAX say, would then have required us to change the message passing programs later. In addition, the communication software to support the communication between computers have been implemented by this time. Thus we have chosen to use two computers in our second version, Version B.

As just described, Version B uses only a single backend. Thus we have converted it to two backends for Version C. This version runs on three computers, a VAX and two PDP-11/44s. However it still lacks several required
functions. There is no concurrency control. In addition all the data, both the database itself as well as the directory, is stored in the primary memory. Thus no disk input and output is required.

By changing from using a simulated disk in Version C to an actual disk system, we have obtained Version D (note that the concept of the build-it-twice strategy is being used to develop a more satisfactory second version of disk operations). This change, though logically simple, has been difficult to implement since it has required us to create a low level interface with the operating system of the PDP-11/44s. Version D includes all the functions we have intended originally for the first real system, Version III, except concurrency control. Thus we have next added a concurrency control process to give us Version E. This version is the same as our original Version III.

The next step in our implementation, Version F, is to change directory management so that directory information is stored on the secondary memory rather than in the primary memory. This change has been more complex than the one required for the actual database since restructuring of the directory data is also required.
As in the original plan, the final version, now Version G, will incorporate access control in the backends and a friendly user interface in the controller or host computer.

Figure 4 shows both the original plan and the revised plan. The versions are identified by the developmental increment. Note the differences and similarities. Versions B, C, and D of the revised plan all concentrate on systems programming issues. The implementation of increased MDBS functionality is delayed until Version E.

3.1.3. A Critique of the DBSIM Implementation Strategy

Overall, the implementation strategy has worked well. The incremental development technique has allowed us to divide the implementation into manageable stages. We have derived satisfaction from the measurable accomplishment of completing a version. Progress on versions B, C and D, after version A has been completed, has been rapid since each successive version represents an increment to the software rather than a new system. Implementation of versions E and F has taken a considerable time. This is attributed to the complexity of concurrency control mechanisms and secondary-memory-based data organizations. We believe that the choice of the incremental development approach has been appropriate
Figure 4: Original and Revised Implementation Plans
given the complexity of the system being implemented and the expertise of the implementors.

By focusing different systems programming issues in different versions, we have also been able to work toward some versions in parallel. For example, the preliminary studies on secondary storage I/O (version D) and secondary-memory-based directory management (version F) have been started before even version A has been completed. The build-it-twice strategy has allowed us to simplify some of the functionality in the earlier versions of MDBS. Repeating the example, we have first implemented a primary-memory-based directory management which is much simpler than a secondary-memory-based directory management. The implementation of a simpler version has given us experience to develop a more satisfactory second implementation. Some people may criticize our application of the build-it-twice strategy (implementing two versions of the same component) by saying that it prolongs the development, especially that the first implementation will be discarded after the second implementation is complete. We, however, believe that it has been necessary to provide the required expertise of the team members. Application of the build-it-twice strategy, along with the incremental development plan, is analogous to
the "stepwise refinement" idea of structured programming. Where as in structured programming we deal with programs, here we deal with large systems.

Application of scaffolding has been useful in training new members. The early implementation of supporting software has resulted in some useful tools. More importantly, it has given us some experience and has reduced the chance of a poor MDBS implementation (lack of experience in implementors could have resulted in a poor implementation).

3.2. The Team Organization

Team organization refers to the makeup of implementation personnel and the level of communication (interaction) between the implementors. The implementation personnel may be grouped into one team, or they may be divided into several teams. If there is more than one team, different teams may be able to work independently or they may need to communicate with each other.

The frequency at which team members change should be considered by the project planners in selecting a team
organization. If the team members are continuously changing, the organization should allow the easy integration of the new members. The expertise of the team members should also be considered in the selection since it can be used toward a more timely development of the system.

3.2.1. Typical Organizations: Democratic Decentralized, Controlled Centralized, and Controlled Decentralized

There are three general team organizations: democratic decentralized, controlled centralized and controlled decentralized [Mant81]. The three organizations differ in team decision making and communication structure in the team.

In democratic decentralized organization (see Figure 5), each team member has equal responsibility and decision-making authority. The group leadership rotates among members based on the necessary skills. The programmer who is most skilled in the current phase of the project assumes the leadership.

The advantage of this type of organization is that a better quality software is produced since the design and code are checked by each team member. The disadvantage is
Figure 5: The Democratic Decentralized Team Organization
the extensive communications among group members. This will result in longer development time. For example, let us say that there are 20 team members. Using the democratic decentralized team organization, the design and code will be reviewed by 20 people. This will improve the quality of the software produced. However, communication among 20 members is high and, most likely, not acceptable. Thus, this organization is practical for small-size teams.

The controlled centralized organization corresponds to the chief programmer team organization described in [Mill71]. In this type of organization (see Figure 6), the team is headed by a chief programmer who has absolute decision-making authority. Other permanent members of the team include a senior-level backup programmer and a librarian. Additional programmers may be added as necessary. The chief programmer does all the design work and writes all of the critical sections of code, for example the routines for subsystem interfaces. The backup programmer is an understudy for the chief programmer, and participates in design and coding; he takes over if the chief programmer leaves the team. The librarian maintains the group's program libraries and coordinates the documentation effort.
Figure 6: The Controlled Centralized Team Organization
One advantage of such a two-level organization is that, since the levels of communication between team members are minimized, development is likely to proceed at a faster pace than with a one-level organization such as the democratic decentralized organization. Also, the system which is developed is likely to be more coherent and consistent, since it was designed primarily by one person. The disadvantage of this type of organization is that the chief programmer is likely to become the bottleneck.

In controlled decentralized organization (see Figure 7), the project manager assigns parts of the work to senior programmers. Each senior programmer supervises a group of junior programmers. The assignment of work to each group (a senior programmer and one or more junior programmers) can be done in two ways. Either each group works on a different module of the system (modular), or it performs a different function, such as coding and testing, of the system (functional).

The modular approach has the advantage of having groups working in parallel, but it would suffer if the work can not be cleanly subdivided into a number of tasks. The functional approach, on the other hand, benefits from
Figure 7: The Controlled Decentralized Team Organization
specialization of talent, but it would suffer from occasional idleness of some functional groups.

3.2.2. The DBSIM Organization: Three-level Chief-programmer-team without Librarians

The MDBS implementation group is organized as a three-level chief-programmer team without librarians. The entire implementation effort is headed by a team supervisor. Separate teams are organized for each subproject being developed; each of these teams is composed of a chief programmer, one backup programmer and one or more programmers. Since the number of implementors is small, group members often serve in different roles on more than one team. An organization chart of the group, depicted in Figure 8, shows four such teams working on the secondary-memory-based directory management, concurrency control, inter-machine communication and disk input/output. This three-level team organization allows different subprojects to be developed in parallel.

We also change the roles of team members during the implementation effort. Thus we are better able to fit our team members into roles that best suits their abilities. Although there is a danger of instability caused by too
Figure 8: The Organization of the MDBS Design and Implementation Teams as of 1/11/83
frequent role changes, this capacity to change roles is especially important to our effort since none of the team members had worked together before the project began. The team supervisor appoints members to the teams and provides overall supervision of the implementation. The team supervisor is permanent, i.e., we do not assign different implementation group members as the team supervisor. A new team supervisor is appointed only when either the current supervisor leaves the project or he does not have the time required to supervise the project.

We have also chosen not to have a librarian for the project. Each group member develops his subproject using his individual account on our computer system. After a subproject is completed, all the documents for the subproject are moved to a special account containing the system documents.

3.2.3. A Critique of the DBSIM Team Organization

The chief-programmer-team organization has been moderately successful. The software architecture of the system has fitted well with this organization. Each process has been developed as a subproject. This has made it possible for each team to work on a reasonable size problem,
and has allowed different components of the system to be developed in parallel. Since the implementation team has been small, project members have often served in different roles on more than one team. This has contributed to the sharing of knowledge among team members. The sharing of knowledge has been helpful since the project members have been changing constantly; when a person leaves the project, the next person assigned to continue his work already has some idea about the subproject, i.e., he does not have to start the subproject from the beginning again. This has avoided loss of effort when a person leaves the project.

We have not employed the concept of teams with the chief and backup programmers in the preliminary studies (such as investigating different secondary-storage organizations for directory management, concurrency control algorithms, and inter-computer communication mechanism). Each of these studies has been done by only one person. The result of this overlook has been that when the original investigator leaves the project, which has happened for all the three studies, the next person assigned to continue his work does not know the subproject as well as a backup programmer would know. This has resulted in a halt in the subproject. A "backup investigator" would have avoided this
problem.

The team organization has allowed different processes to be developed in parallel. We have not, however, specified the interfaces between the processes before developing the processes. Our failure in this respect has necessitated modification of some processes completed earlier. For example, after the directory management process in the backends has been developed, we have had to modify it somewhat to provide some information required in the record processing process in the backends.

3.3. The Software Quality Assurance (SQA)

The major concern in SQA is to ensure that the end product satisfies its requirements. The end product should also be well structured in order to make the maintenance of (and extension to) the system possible.

The project planners need to select techniques for assuring the quality of the software produced by team members. Team members must be supervised to assure that they employ the techniques and that the software produced by them is well structured.
3.3.1. Typical SQA Techniques: Design and Code Inspections, Reviews, Audits, and Structured Walkthroughs

The typical SQA techniques suggest the same procedure: review the content of a document for quality and consistency. The characteristics (simplicity, reliability, understandability and maintainability) used in determining the quality of a system have been described in Section 2.1.5, so we do not repeat them here. The consistency of a system indicates that the assumptions and assertions in different modules of the system are not contradictory. For example, the input/output specification for a module should correspond to the invocations of the module.

A widely used SQA technique is structured walkthrough [Your79a]. A structured walkthrough is a formal review of the software development effort at a given stage in its development cycle. The work is reviewed by a walkthrough committee, with the purpose of finding any errors that may be present. The purpose of a walkthrough is not to solve problems, only to identify them; neither is a walkthrough a management tool to evaluate any employee's performance.

Each member of the walkthrough committee has a well-defined role. A coordinator organizes and runs the meeting.
The presenter, the originator of the work, presents his work to the group and answers questions. The reviewers examine the material before the walkthrough is held, and, during the walkthrough, present their findings. A scribe records the proceedings. Each member votes on the outcome; the material may be accepted as presented, accepted with revision, or returned for revision and subsequent walkthrough. If major changes, such as fixing a design error, to the material presented are required, the material will be returned for revision and subsequent walkthrough. If only minor changes, such as changing a variable name to a more meaningful name, are required, the material will be accepted with revision. Finally, if no change is required, the material will be accepted as presented.

3.3.2. The SQA Techniques Used in DBSIM: Formal Reviews for Design and Code

The MDBS teams have used the structured walkthrough technique both at the design stage and at the coding stage. All detailed design specifications and source code are reviewed by walkthrough committees. These committees are chosen to include members from more than one chief-programmer team. This practice contributes to a more
effective walkthrough since not all participants are involved in the development of the material being reviewed. It is also valuable in cross-training team members in areas other than those to which they are assigned. The status of a task can be determined by reviewing the walkthrough reports for that task. Figure 9 shows a sample walkthrough report.

3.3.3. A Critique of the SQA Techniques Used in DBSIM

The use of structured walkthroughs has been very effective. It has helped us to identify many design and program code problems. It has also proved to be a very useful way of sharing knowledge, which is necessary given the changing composition of the implementation team. We feel that we have become quite skillful in practicing this technique.

We have also experimented with informal code reviews as a precursor to formal code reviews in an effort to expedite the process. The participants in an informal code review are only the programmer and the designer of a part. The participants in a formal code review, as in other formal reviews, are the originator of the work (the programmer) and at least three reviewers. The idea behind informal code reviews is to find minor mistakes in the code more
WALKTHROUGH REPORT

Coordinator: M. Higashida
Project: MDBS Record Processing

Coordinator's Checklist:
1. Confirm with producer that material is ready and stable.
2. Issue invitations, assign responsibilities, distribute materials.

DATE Apr. 15, '92    PLACE CA 230
TIME 14:30    DURATION 30 Minutes

<table>
<thead>
<tr>
<th>Participant</th>
<th>Role</th>
<th>Can Attend</th>
<th>Has Material</th>
<th>Initials</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ali Orosji</td>
<td>Reviewer</td>
<td>✓</td>
<td>✓</td>
<td>A.D.</td>
</tr>
<tr>
<td>Paula Strauss</td>
<td>Scriber</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>He Xingyi</td>
<td>Presenter</td>
<td>✓</td>
<td>✓</td>
<td>H.D.</td>
</tr>
<tr>
<td>M. Higashida</td>
<td>Coordinator</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Agenda:
1. All participants agree to follow the (same) set of rules.
2. New project: walkthrough of material.
3. Old project: item-by-item checkoff of previous action list.
4. Creation of new action list (contributions by each participant).
5. Group decision.
6. Deliver copy of this form to project management.

Decision: ✓ Accept product as-is
✓ Revise (no further walkthrough)
               Revise and schedule another walkthrough
(Participants should initial above.)

Figure 9: A Sample Walkthrough Report
efficiently; since only two people are involved in an informal code review, as opposed to 4-5 people in a formal code review, manpower is saved. A code is not, however, approved until walkthroughs with at least three participants are scheduled and the reviewers approve the code.

One general concern for software engineers is whether or not to employ reviews since there is not sufficient statistics on errors found at reviews versus the time and effort needed for reviews. Our estimates, from reviewing our walkthrough reports, are shown below
Number of walkthroughs: 63

<table>
<thead>
<tr>
<th>Number of errors found at a walkthrough</th>
<th>Percentage of the number of walkthroughs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.12</td>
</tr>
<tr>
<td>2</td>
<td>17.94</td>
</tr>
<tr>
<td>3</td>
<td>21.79</td>
</tr>
<tr>
<td>4</td>
<td>23.07</td>
</tr>
<tr>
<td>5</td>
<td>14.10</td>
</tr>
<tr>
<td>6</td>
<td>6.41</td>
</tr>
<tr>
<td>&gt;= 7</td>
<td>11.53</td>
</tr>
</tbody>
</table>

Average number of errors found at a walkthrough: 4.05
(Note: This error count includes only the major errors pointed out by the reviewers, i.e., it does not include the minor errors.)

Average elapsed time for a walkthrough: 50 minutes
(Note: This time does not include the time that reviewers spent before a walkthrough to read the material to be presented/reviewed in the walkthrough.)

We believe that our benefits from employing reviews have far cost-justified the time and effort. In summary, the use of reviews has helped us to

(1) identify almost all the design errors and most of the code errors,

(2) share knowledge, and

(3) control the quality of the design and code.
3.4. The Need for Periodic Overall System Reviews

We have described the management techniques used in our implementation effort in the preceding sections. Let us now describe one problem that has occurred in our implementation effort and that has not been detected by any of our management techniques. We will first describe the problem in Section 3.4.1. We then, in Section 3.4.2, suggest an additional management technique that might have prevented the problem from occurring.

3.4.1. The Problem with Concurrency Control

Concurrency control refers to the mechanisms that allow concurrent and interleaved execution of user requests while preserving the integrity and consistency of the database. (The integrity of the database indicates that the values in the database satisfy certain conditions. For example, the salary of an employee should be greater than zero. The consistency of the database indicates that the values in the database are not contradictory. For example, if the salary of an employee is stored in two different places in the database, the two values should be the same.) Access to the directory data as well as access to the user data may have to be controlled in order to provide correct results.
(responses) to the database users.

We have started the MDBS implementation using the requirements specification presented in [Hsia81a, Hsia81b]. These documents describe the concurrency control mechanism for controlling access to the user data in MDBS. The tables for storing the directory information are also described in these documents. Details on the implementation of the directory tables are left to the implementors. Thus, these documents have not addressed the concurrency control issue of updating the directory data in MDBS. We, the implementors, have not originally realized that we need to investigate the issue either. As a result, the issue of whether or not concurrency control is needed for the directory data (if concurrent and interleaved execution of requests is to be allowed) has been originally overlooked.

Concurrency control is needed in MDBS for the directory data as well as the user data (this is described in detail in Chapter 5). However, we have not realized the need for the directory-data concurrency control until almost one and one half years after the implementation effort has started. This late detection has caused us to modify the code, which has already been completed, for the processes in the
backends. Some processes, such as record processing, have needed minor changes. Some processes (such as directory management) have, however, needed major changes. This has resulted in loss of effort and it has delayed the completion of the system.

One reason for this late detection is that the issues of concurrency control implementation have not been addressed until version E, i.e., we have not investigated the impacts of concurrent request execution until version E of MDBS (this is the original plan which is based on the incremental development strategy). The main reason for this late detection may, however, be the lack of a procedure in our management techniques for "periodic overall system review." This concept is discussed further in the following section.

3.4.2. The Concept of Overall System Reviews

It is useful to review the system periodically to assure that there are no inconsistencies or problems in the system as a unit (by system, we mean the final end product, i.e., all the components that have been implemented and all the components that need to be implemented). The idea is that as the implementation progresses, the system is
understood more clearly. Thus, a review of the system may detect problems in the system not seen in the earlier stages. These reviews should be done periodically, as opposed to being done only at the beginning and/or at the end of the implementation effort. A system review at the beginning of the implementation effort would find some errors, but not necessarily all the errors. A system review at the end of the implementation effort would find all the errors, but major changes to the system would be required to fix the errors (since the errors are detected too late). The periodic system reviews would find errors as early in the implementation effort as possible, so it would be less costly to fix the errors. The periodic system reviews should, of course, include reviews at the beginning and at the end of the implementation effort. Application of periodic overall system reviews in our implementation effort might have detected our problem with concurrency control earlier and saved some time.

The project planners should schedule overall system reviews. The chief programmers in implementation teams should participate in these reviews. An outsider who is not involved with the detailed design of the system may also be helpful for these reviews since he is not "brain washed"
(like the implementors!) with the low-level details of the system. The overall behavior and operation of the system should be studied in these reviews in an effort to find inconsistencies and problems in the system.

In an overall system review, a reviewer is not concerned with the mechanical details for the individual system components, i.e., low-level details are ignored in such a review. The reviewer is, rather, concerned with the whole system at a higher level. This review is likely to find inconsistencies and problems in the system in the early stages of the development effort.

3.5. Concluding Remarks on the Management Techniques

The incremental development and build-some-twice-different-versions strategy has been effective. The incremental development technique has allowed us to build the system in manageable stages. The build-it-twice strategy has allowed us to simplify the earlier versions of MDBS and gain experience. The development of simplified versions of some components (such as the I/O subsystem) has made it possible to test other functionalities of MDBS in a simpler system. We believe that we could not have developed
the complete system at once because we have had no systems programming experience and knowledge of operating systems prior to the implementation effort (by lack of knowledge in operating systems we mean lack of experience in using operating system facilities such as low-level I/O operations and message-passing mechanisms).

The team organization has allowed different components of the system to be developed in parallel. This has made better use of the manpower and has reduced the development time. The reviews of the design and code by several people have found almost all the design errors and most of the coding errors. This has greatly reduced the testing effort. The suggestions and comments made by the reviewers at the walkthroughs has also resulted in a well-designed and well-structured end product.

Finally, we now realize that the entire system should be reviewed periodically in order to find inconsistencies and problems in the system at their earliest stages.
4. SOLUTIONS TO THE ISSUES OF SYSTEM HARDWARE AND SOFTWARE

The issues of system hardware and software have been introduced in Section 2.2. First, hardware for different system components must be chosen. Second, a systems programming language must be selected for coding. Third, operating systems for the development time and run time must be chosen. Finally, effective software development techniques must be selected for different phases of the software life cycle (requirements specification, design, coding and validation). In this chapter, we describe the alternative solutions to these issues that the MDBS implementors considered. The solutions adopted, the rationale for choosing them, and the results of our experience are also given.

4.1. The System Hardware: VAX-11/780, PDP-11/44, and PCL

Project planners want to select hardware which satisfies the requirements of the hardware organization at
the smallest price. The hardware organization for MDBS has been shown in Figure 2. That organization assumes that the controller and backends are connected by a broadcast bus. It also assumes that the dedicated disk drives at the backends have the capacity to support very large databases.

Since the MDBS project planners have anticipated an equipment grant from the Digital Equipment Corporation (DEC), a proposal for DEC equipment is drafted. The proposal suggests that the most cost effective selection of hardware would be PDP-11/34s for the backends and a PDP-11/70 for the controller. At the time this proposal is drafted (winter of 1979), the latest generation of the corresponding DEC minicomputers is represented by the PDP-11/44 and the VAX-11/780.

In MDBS, performance is improved by increasing the number of backends in the system. MDBS is designed to be easily extensible so that no software development or downtime costs are incurred in expanding the system. The expense incurred will be the cost of the hardware. Therefore, the cost of adding backends to the system is an important measure of cost effectiveness. In 1979, the PDP-11/34 minicomputer is the least expensive model in the PDP-
11 series which supports large-capacity, hard disks and can be interconnected with DEC's Parallel Communication Link (PCL). Using PDP-11/34s as backends will minimize expansion costs for MDBS. Hardware cost is less important in selecting the controller than in selecting the backends. The PDP-11/70 (the most powerful PDP-11 computer) can assure that the controller will not become a bottleneck in MDBS, even though most of the work has been delegated to the backends. The PDP-11/70 is also supposed to be used during the development effort. The additional computing power of this machine can reduce the development time.

DEC's response to the original proposal is that since the ultimate goal of the implementation effort is database systems research and not product development, the latest technology available should be used. Although newer equipment may be more costly, it may also enhance the research and implementation effort. The final agreement, therefore, shows: PDP-11/44s are used as the backends, and a VAX-11/780 is used as the controller and to support the program development effort. Since a broadcast bus hardware such as Ethernet [Metc76] required by the MDBS design is not available, PCL is used for the purpose of simulating the broadcast and parallel transfer capabilities. The PCL is
used to interconnect the VAX-11/780 and PDP-11/44s. The impacts of this replacement hardware will be discussed in Section 5.3.

4.2. The Systems Programming Language

A systems programming language for the MDBS implementation must be powerful yet relatively easy to use. In other words, the language must have enough constructs to program the features for the multi-backend database system. It is also important to choose the programming environment and language constructs which will make the development effort easier. The implementation team for MDBS is composed primarily of computer science students who have little practical experience, although they have a broad base in textbook knowledge. A systems programming language which makes the development effort easier will help these relatively inexperienced implementors to develop more reliable software.

4.2.1. Typical Languages: Bliss, C, and MAINSAIL

Systems programming languages can be evaluated in terms of: availability, programming environment, language
features, portability, efficiency and reliability. The portability, efficiency and programming environment factors of a language have been described in Section 2.2, so we do not repeat these definitions here. The availability of a language is determined by the number of computing systems and operating systems that can be used for compiling and running the programs written in the language.

The language features are the constructs in a programming language. The language used in the implementation of a system should have enough constructs to program the required functions in the system. The constructs should also be easy to use, easy to read and easy to understand.

The reliability of a language involves whether or not the instructions in a language actually do what they are purported to do by the language designers and compiler writers. Clearly an unreliable language leads to unreliable software. The reliability of a language is improved when the language has type-checking mechanisms. Such a mechanism assures that the data types of the operands in an expression (or subexpression) are compatible with the operation which is to be performed.
The MDBS project planners have examined three systems programming languages: Bliss, C, and MAINSAIL. A brief evaluation of each language is given in the following sections.

4.2.1.1. The Bliss Language and Its Compilers

The Bliss language [Wulf71] is originated in the Department of Computer Science at Carnegie-Mellon University. Availability of Bliss is rated high since this language is provided by DEC for PDP-11 and VAX systems. There are, however, different versions (dialects) of Bliss and there are significant differences between them, so the portability is low. Another limitation is that object code for the PDP-11 must be generated by a cross compiler running on a larger computer system.

There is no set of programming tools for Bliss programmers, so the programming environment is poor. Bliss is an expression-level language. In its syntax, all identifiers denote addresses rather than values, so a de-reference operator ('.') must be used. This notation makes it difficult for the programmers who have not used the language before to write or read Bliss code. The language supports no primitive data type. Since operators are never
type-specific, type-checking is non-existent. Nevertheless, an advantage of Bliss is that it supports the data abstraction concept.

There is some question as to the reliability of Bliss, since it contains so many low-level features. It does, however, seem to be the best of those languages surveyed when measured in terms of time/space efficiency on the DEC equipment.

4.2.1.2. The C Language and Its Programming Development Environment

UNIX [Ritc74] is a Bell Telephone Laboratories trademark, and UNIX operating systems are licensed by Western Electric. Although the C language [Kern78] is originally designed for and implemented with the UNIX operating system for the DEC PDP-11, it is not tied to any particular operating system or architecture; C compilers are available on many systems. Not all versions of C are compatible, so portability can be a problem. C is supported with all versions of UNIX, and is available from the Digital Equipment Computer Users Society (DECUS), for use with operating systems for PDP-11 and VAX.
A rich set of program development tools accompanies UNIX system software. These tools provide a very good environment for C programmers. C syntax is very simple. The language supports primitive data types such as integer and characters; type-checking, however, is not strongly enforced. C compilers usually do not support extra features, such as sophisticated macroprocessing, but many of these features are available in the programming environment support provided with UNIX. C, unlike Bliss, does not support the data abstraction concept.

C is reasonably reliable, even though many vendors do not commercially support the language. It is also reasonably efficient. A good textbook for C users is available [Kern78].

4.2.1.3. The MAINSAIL Language and Its Relationship to the other Languages

MAINSAIL (MAchine INdependent SAIL) [Wilc77] is evolved from the programming language SAIL, which is developed at Stanford University's Artificial Intelligence Laboratory. XIDAK, Inc. owns exclusive rights to develop and market MAINSAIL. The language is distinguished by its portability. The same compiler and run-time system, both written in
MAINSAIL, are the basis for every implementation; code generators and procedures which interface to the operating system must be specially written. MAINSAIL is implemented for DEC PDP-11 systems.

MAINSAIL is developed and is marketed with a set of integrated tools for program development. The syntax of the language is similar to ALGOL-60. Consequently, it appears familiar to most people with formal training in computer science. The set of data types supported is more extensive than that supported either by Bliss or by C, and there is strong type-checking. On the other hand, the major disadvantage of the language is that there is no capability to invoke subroutines written in a language other than MAINSAIL or assembly language.

Reliability is rated good. Efficiency, however, is rated lower than that of either Bliss or C. Low-level features must be coded in an assembly language, which implies that two development languages must be learned rather than one.
4.2.2. The Language Used in MDBS Implementation: C

The efficiency ratings of MAINSAIL and the requirement that low-level features are to be coded in an assembly language quickly eliminate that language from consideration. The real choice, then, is either Bliss or C. C has a number of features which make it more desirable than Bliss.

First, C has fewer language constructs and it has a simpler syntax. Given the inexperience of the implementation team, it is important to choose a language which can be quickly learned. The simple syntax of C makes this language very easy to learn. Furthermore, a good textbook for C users is available [Kern78]. Next, the programming environment which can be provided under UNIX for developing C programs is a major consideration. A third factor is that the compilation process for Bliss would require the resources of a computer system at considerable distance from where the MDBS implementation effort was underway (the Laboratory for Database Systems Research). The laboratory initially had only two PDP-11/44 systems, either of which is not large enough to support the Bliss compiler. C, then, is the language we have chosen, since it can make the greatest contribution toward the goals of the
implementation effort.

4.2.3. A Critique of the Language Used in MDBS Implementation

Overall, using C has been an appropriate choice and it has worked well. The simple syntax of this language is easy for the implementors to learn. The programming environment provided under UNIX for developing C programs has also been very helpful. We have, however, encountered some problems in using the C language [1]. We have found that multiply-defined variable/subroutine names are apt to surface when integrating program segments written by different programmers. In some measure this is due to the rules for uniqueness in names in the C language. Variable names must be unique within the first eight characters; function names must be unique within the first six characters. It is also in part a product of scope rules. Function names are always external variables within a file, i.e., function names are global within a file. (The block-oriented languages, such as PASCAL, allow internal subroutines in a program. This

[1] The problems discussed here pertain to the C compiler that we have used (C compiler on RSX). More recent C compilers have solved some of these problems.
greatly reduces the problem of multiply-defined names.)

Another difficulty inherent in the language is that, since the C language supports functions and not procedures, all simple parameters are value parameters. In order to pass a simple parameter to a function by reference (instead of by value), a pointer to the parameter must be passed to the function. Almost inevitably, an application arises where one must pass a pointer to a pointer in order to return multiple values from a function. This does not lead to easily comprehensible programs. Finally, uninitialized variable references and out-of-bound array references are not detected by the C compiler and run-time support system. It would be helpful to have a compiler which includes these features for program development. Such features would greatly reduce program testing effort.

4.3. The Development-Time and Run-Time Operating Systems

The important consideration in choosing a development-time operating system is the suitability of the operating system features for the development effort. The important considerations in choosing the run-time operating systems are system performance and suitability of the operating
systems features for the MDBS application. Suitability for MDBS application is related to the requirements of MDBS, e.g., the underlying operating systems must be able to support multiple processes. The development-time and run-time operating systems need not be the same so long as the software produced on the development-time operating system is portable to the run-time operating systems.

4.3.1. The Operating Systems Used in MDBS Implementation: UNIX, RSX, and VMS

An operating system which is easier for inexperienced programmers to use will be more suitable for MDBS development. UNIX [Ritc74] is a user-friendly operating system (some naive users may find UNIX hard to use at the beginning because of its abbreviated (cryptic) commands; but after one or two weeks, they will find the system easy to use). Interactive programs which teach the user how to use operating system facilities are a part of the UNIX package; all documentation is available on-line. A variety of aids to C programmers are available. An example is the program LINT [John78a], which checks C programs for errors, such as type violations, which are not checked by the C compiler. Other software tools, which are provided by UNIX, include
MAKE [Feld78], YACC [John78b], LEX [Lesk79] and GREP (the usage of these tools is described in Section 4.3.2).

The characteristics mentioned above make the UNIX environment desirable for program development. UNIX does, however, lack some system features which are required for efficient implementation of MDBS. For example, the message-passing mechanisms in UNIX are not satisfactory for MDBS implementation. Message-passing between processes (in UNIX) is done using "pipes" (the new version of UNIX, which was not available when we started the MDBS implementation, provides new message-passing mechanisms). Pipes can not be used for arbitrary exchange of messages between processes, which is required in MDBS. Thus, UNIX is used only during the MDBS development. The underlying operating systems for MDBS at run time are VMS on the VAX and RSX-11 on the PDP-11s. These operating systems provide features, such as efficient message-passing mechanisms, required in MDBS.

The performance of the run-time operating system for the backends is a major factor on the overall performance of MDBS. RSX-11 is a DEC real-time operating system. Since real-time operating systems are engineered for execution speed, RSX-11 is desirable from the performance standpoint.
RSX-11 also provides more flexibility than UNIX; implementors can choose which operating system features to use, i.e., the features included in the operating system (RSX-11) used for MDBS operation time can be a subset of the standard features of RSX-11, in order to save disk and memory space.

The performance of the run-time operating system for the controller is not as crucial as the performance of the run-time operating system for the backends since most of the work, especially operating system related work such as disk I/O, is done by the backends. An efficient operating system is still desirable for the controller since it can improve the overall system performance. VMS provides an efficient message-passing mechanism which is required in MDBS, so this operating system is used for the controller at run time.

4.3.2. A Critique of the Operating Systems Used in MDBS Implementation

We feel that the choice of UNIX as a development environment has been a good one. Several software tools available on UNIX have been helpful. We have used LEX [Lesk79] to develop the lexical analyzer for MDBS requests. LEX is a lexical-analyzer generator which can be used to
generate programs in C. The input to LEX is a specification of the tokens of the language (the tokens of the MDBS data manipulation language, DML, for our case) in regular expression form, and subroutines which specify the actions to be taken upon recognition of the tokens. LEX generates a program in the C language. This program includes a representation of a deterministic finite-state automaton generated from the regular expressions of the source, an interpreter which directs the control flow, and the subroutines from the source. The lexical analyzer produced by LEX is easily interfaced with the parser generated by YACC.

We have used YACC [John78b] to develop the request parser in MDBS. YACC, yet another compiler-compiler, has been used to generate a parser which calls the LEX-generated lexical analyzer for tokens, and organizes the tokens according to rules of a grammar. When a rule is recognized, some specified action is taken. The input to YACC is a specification which includes declarations of token names, the rewriting rules of the grammar, and action programs. YACC produces a C program, the parser, according to the specification. The parser operates like a finite-state automaton with a stack. The top-of-stack represents the
current token. The parser also has access to the next token, called the lookahead token. Using this simple mechanism, the parser can determine whether the input DML statements are syntactically correct.

We have used LINT [John78a] to find a lot of errors, such as missing parameters and type violations, not detected by the C compiler. LINT is a command which examines C source programs, detecting a number of errors not detected by the C compiler. It also detects a number of wasteful, or error prone, constructions such as unused variables. We have used MAKE [Feld78] to build the overlay structure and for compiling different processes. In a programming project, it is easy to lose track of which files need to be recompiled after a change is made in some part of the source. MAKE provides a simple mechanism for manipulating up-to-date versions of programs. The input to MAKE is a list of dependencies. Each dependency indicates which files are required to be up to date before a specific file can be processed (e.g., be compiled). Whenever a change is made to any part of the program, the MAKE command (using the list of dependencies) will compile all the files that need to be compiled as a result of the change.
GREP is a pattern-matching command. This command has been useful for locating, for example, all the places a variable occurs. This command has been used when we need to search a set of files at once (an editor can only search one file at a time). We have also used this command to collect statistics on the source code (the statistics are given in Chapter 6). We feel that the use of software tools available on UNIX has greatly reduced the implementation effort.

RSX-11M and VMS provide system features, such as message-passing mechanisms, which are required for implementation of MDBS. Therefore, we have used facilities provided by these operating systems, instead of writing our own. This has reduced the implementation effort. We should have, however, chosen RSX-11M-PLUS instead of RSX-11M. This operating system allows separate instruction and data spaces. Thus, the program address space under RSX-11M-PLUS is twice as much as that under RSX-11M (128 kilobytes vs. 64 kilobytes). The larger program address space of RSX-11M-PLUS reduces the problems with overlaying a process. We have not, however, originally known this difference. Our lack of knowledge in operating systems has, in general, been a major problem (by lack of knowledge in operating systems
we mean lack of experience in using operating system facilities such as low-level I/O operations and message-passing mechanisms).

4.4. Software Development Techniques

These are the techniques used in different phases of software life cycle (requirements specification, design, coding and validation). Effective software engineering techniques should be used when developing a large system to avoid software crisis (cost, schedule and quality) [Boeh73]. Without effective techniques, a development effort will be too costly and/or fail to produce reliable software. In the following sections, we describe techniques proposed by different software development methodologies, and the techniques used in DBSIM. The results of our experience in using the techniques are also given.

During the requirements specification of the software life cycle, a functional description of the system is produced. That is, "what" the system does is specified during this phase. We have begun MDBS development using the requirements specification presented in [Hsia81a, Hsia81b]. Thus, we will not discuss the techniques and tools for the
requirements specification phase. We will, instead, concentrate on the design, coding and validation phases of the software life cycle.

4.4.1. Design

During the design phase of the software life cycle, an algorithm (method) is devised to carry out the specification produced in the previous phase. That is, "how" to implement the system is specified during this phase.

4.4.1.1. Typical Techniques: Top-Down, Bottom-Up, and Data-Structure-Driven

There are three basic design strategies: top down, bottom up and data structure driven. In the top-down design strategy, each problem is successively decomposed into smaller problems (subproblems). The decomposition process is continued until the problems at the lowest level can be easily implemented (coded).

The main advantage of top-down design is that the designer can concentrate on one component of the problem at a time. The disadvantage is that it is possible to get multiple solutions for the same problem, since a problem can be decomposed into subproblems in more than one way.
In the bottom-up design strategy, the lowest level of the system is developed first. Each higher level is then designed using the services provided by its lower level.

The main advantage of bottom-up design is the concept of data abstraction inherent in this strategy (each level is considered an abstract machine that provides some services to its higher level). The disadvantage of bottom-up design is that in most cases it is not obvious where to begin or what the capabilities of each level should be.

Top-down and bottom-up design strategies are based on the functionality of the system being developed, i.e., they concentrate on what has to be done. The data-structure-driven design strategy, on the other hand, is based on the input and output of the system, i.e., it concentrates on defining the structure of the input and output. Once these structures are defined, the program that manipulates them is developed.

The advantage of data-structure-driven design is that, since data structures are usually well defined and the program structure is determined by the data structures, most people using this strategy will come up with similar program structures. There are, however, some practical problems
with this strategy. For example, erroneous data do not have a particular structure, so error processing can not be integrated in program structure in a structured way.

4.4.1.2. The DBSIM Techniques

During the design stage of the software development cycle, the detailed program specification is developed. A design strategy is selected first. Then a design methodology with which to implement the strategy is chosen. A top-down design strategy, implemented in a systems specification language, is used in the MDBS implementation effort. This choice is argued and described in the following paragraphs.

(A) Top-Down Design and Use of Data and Service Abstractions

A top-down design strategy is a natural choice for MDBS. The design and analysis study in [Hsia81a, Hsia81b] clearly describes the top level design. It also suggests the possibility of functional decomposition, i.e., the entire system can be broken into discrete functional units. For example, the execution phases of a retrieval request can be broken down into directory management and record processing. Directory management, an example of a
functional unit, includes three phases: descriptor search, cluster search, and address generation. (For precise definitions of these terms and phases, the reader may refer to [Hsia81a, Hsia81b, He83, Kerr83, Boyn83a, Boyn83b, Demu84].)

The stepwise-refinement concept of the top-down design strategy is effective on all the functional units in MDBS. As another example, record processing is decomposed into four types of request processing: insert processing, retrieve processing, delete processing and update processing.

At a lower level, the concept of data and service abstractions, which originates in the bottom-up design approach, is used. Since MDBS is being developed as a prototype system for research into performance evaluation, we anticipate that data structures and system services will be routinely modified in attempts to measure the effect of different data structures on system performance. The abstractions allow us to separate the basic system functions from the data structures and from the implementation of the services, minimizing the effect on the system when data structures or implementation services are modified.
All the data structures, such as the directory tables, have been implemented using data abstractions. All the services required by different functional units in MDBS have been implemented using service abstractions. For example, we have implemented procedures for performing I/O operations and made them available to directory management and record processing. As another example, we have implemented procedures for sending/receiving messages and made them available to functional units in the controller and backends.

(B) A Systems Specification Language (SSL)

The design methodology which the MDBS implementation group uses is a systems specification language (SSL) modeled on the process design language (PDL) described in [Ling79]. The SSL is characterized by a number of constructs for the expression of the different levels of a system: system, subsystem, module and procedure. A system is at the highest level of the hierarchy. MDBS, in this case, is the system.

At the second highest level of the hierarchy, we have the level of subsystem. A subsystem is a separate component of a system. In other words, each system may consist of several subsystems. The MDBS controller, for example, is a
subsystem, as is each MDBS backend. The system, consisting of the controller and the backends, is the MDBS.

Below the level of subsystem, we have the level of module. A module is intended for the implementation of a data or service abstraction. It consists of the procedures and data structures implementing the abstraction. A procedure is at the lowest level of the hierarchy. It corresponds to the usual notion of a subroutine. Procedures are invoked to perform some work on some input data and produce some output. However, they are not allowed to retain data between invocations. An example of a module (and its procedures) is the attribute-table module. This abstraction is used to implement one of the directory tables in directory management (the directory tables are described in Chapter 5). The data structure for this module (abstraction) is the attribute table. The procedures in this module perform operations such as adding an attribute to the table and returning the information on an attribute in the table.

A formal outer syntax and an informal inner syntax are used in a procedure. The outer syntax allows only the following three types of constructs: sequence, decision and
iteration. Below is an example of the if-then-else decision construct.

    if expression
    then
        statement sequence
    else
        statement sequence
    end if

The underlined words represent the formal outer syntax. The other words represent the informal inner syntax; the only requirement for this inner syntax is that it must be understood by all project members. Figure 10 shows a typical SSL procedure specification.

The use of a systems specification language has been helpful in

(1) communication among team members,
(2) reducing dependence on individuals, and
(3) producing complete and accurate documentation on the design.

(C) The Use of Jackson Charts

Our original designs have been developed using only SSL. After using SSL for a few months, we have realized that a two-dimensional representation of the program structures would make it easier to understand the design.
4.10.21.1 proc LIST_TYPE_C_ATTNAMES /* TYPECLST (DBL1U3J) */
        (input: type_C_attr_names,
         atpointer);
        /* List all the attribute names over which type-C descriptors */
        /* are to be defined. Input is a list for attribute names */
        /* over which type-C attributes are to be defined, and a */
        /* pointer to the AT. */

4.10.21.2   scalar index, /* index to list of attribute names */
            attr_name,
            duplicate, /* indicator - TRUE or FALSE */
            dditpointer,/* pointer into DDIT returned from ATM*/
            descr_type; /* A, B, C, or NOTFOUND */

4.10.21.3   index := 1;
4.10.21.4   type_C_attnames[index] := null; /* null indicates end
of list */
4.10.21.5   while more type-C descriptors do
4.10.21.6   get attr_name from terminal;
4.10.21.7   perform ATM$FIND(attr_name,
4.10.21.8   dditpointer, /* null Indicates end
4.10.21.9   of list */
4.10.21.10  index := index + 1;
4.10.21.11  type_C_attr_names[index] := null;
4.10.21.12  if a type-A or type-B descriptor is already defined
4.10.21.13  over this attribute name
4.10.21.14  then
4.10.21.15  display error message;
4.10.21.16  if duplicate is FALSE:
4.10.21.17  then
4.10.21.18  type_C_attnames[index] := attr_name;
4.10.21.19  index := index + 1;
4.10.21.20  end if
4.10.21.21  end while
4.10.21.22 end proc

This number means that this is the 22nd
program statement in this procedure. The
procedure number is 4.10.21 which means that it
was called at program statement 21 in the level-3
procedure numbered 4.10. That procedure was, in turn,
called at program statement 10 of the level-2 procedure
numbered 4. Procedure 4, in turn, was called by program
statement 4 in the main procedure.

Figure 10: An SSL Specification of a Program Procedure
Thus, since January of 1982, we have used a technique, Jackson charts [Jack75], to represent the program structures. We have chosen Jackson charts, rather than flowcharts, since they reflect the structure of the design better and are more helpful.

Three constructs are used in a Jackson chart:

(1) Sequence - Figure 11a shows a sample sequence. In this example, the sequence A consists of B followed by C followed by D.

(2) Iteration - Figure 11b shows a sample iteration. In this example, the iteration A consists of multiple occurrences of B.

(3) Selection - Figure 11c shows a sample selection. In this example, the selection A consists of one of B, C or D.

A sample program structure and its corresponding SSL are shown in Figures 12 and 13, respectively.

Jackson charts contain fewer details than the SSL specifications. The SSL specifications contain the information present in Jackson charts plus variable declarations and procedure calls, i.e., Jackson charts and the SSL specifications with variable declarations and procedure calls removed are basically the same. Jackson charts, however, provide a two-dimensional representation of the program structures. A two-dimensional representation
Figure 11: The Constructs Used in a Jackson Chart
Figure 12: A Sample Program Structure
10.1 proc DELETE_PROCESSING(input: QUERY, ADDRESSES);
    /* This procedure is used for processing DELETE requests. */

10.2 list QUERY : string;
10.3 set ADDRESS : integer;
10.4 array TRACK_BUFFER : word;
10.5 scalar indexA, indexB : integer; /* these are pointers for ADDRESSES and TRACK_BUFFER respectively */
10.6 scalar satisfied : logical;

    /* process data track by track */
    for each address ADDRESS(indexA) in ADDRESS do
        /* fetch one track into TRACK_BUFFER */
        perform FETCH_TO_TRACK_BUFFER(indexA, TRACK_BUFFER);
        /* select records in TRACK_BUFFER one by one */
        for each record TRACK_BUFFER(indexB) in TRACK_BUFFER do
            if the record is not marked for deletion
                /* check whether the record satisfies the QUERY */
                perform CHECK_QUERY(QUERY, TRACK_BUFFER, indexB, satisfied);
                if satisfied = TRUE
                    then /* mark the record in TRACK_BUFFER(indexB) */
                        perform DELETE(TRACK_BUFFER, indexB);
                        /* for deletion */
                end if
            end if
        end for /* indexB */
        /* store TRACK_BUFFER back to disk */
        perform STORE_TRACK_BUFFER(indexA, TRACK_BUFFER);
    end for /* indexA */
end proc

Figure 13: The SSL Corresponding to the Sample Program Structure in Figure 12
reflects the structure of the program better and it is easier to understand ("a picture is worth a thousand words"). Jackson charts, along with the SSL specifications, have been used to document the detailed design of MDBS.

4.4.1.3. A Critique of the DBSIM Design Techniques

Our design approach has been effective, and has fitted well with the implementation strategy. Application of the concepts of abstraction and information hiding has been particularly valuable. The implementation of key parts of the MDBS system as data abstractions has allowed us to progress from version to version by replacing code at the low level, with no changes to the higher levels of the system. This is particularly true where we have adopted the build-it-twice approach. The secondary storage I/O subsystem and the directory management subsystem have been designed and implemented using both data and service abstractions.

The use of a systems specification language has been very effective. The constructs of the formal outer syntax are familiar to students of computer science. The informal inner syntax allows a great deal of flexibility. The result is a specification which is precise and easily readable.
This is helpful for communication among members. It also reduces dependence on individuals; when a person leaves the project, the next person assigned to continue his work has all the details on the subproject, i.e., he does not have to start the subproject from the beginning again. This avoids loss of effort when a person leaves the project.

We have set no firm standard for the level of detail to which the specification is to be developed. This has been left to the judgment of the reviewers. An unexpected side-effect is that, where specifications are given in considerable detail, coding is often a direct translation from the SSL into the C language. This has created difficulties where mechanical details such as checking that subscripts are within array bounds are not included in the design, and are overlooked by the programmers. Mechanical details are compiler and run-time-support dependent. For example, some compilers produce code for checking array references; some compilers do not. Thus, mechanical details should not be included in the specifications for the design. The programmers should, however, be cautioned to include these details when writing programs, and the reviewers should check this in code walkthroughs.
We have noticed that there are times when we need a two-dimensional representation of program structure. Since January of 1982, we have used Jackson notation [Jack75] in program structure charts. Jackson charts have less details than SSL and they make it easier to understand the design. These charts, along with the SSL specifications, have been used to document the detailed design.

4.4.2. Coding

During the coding phase of the software life cycle, code in some programming language is generated from the design specification developed in the design phase.

4.4.2.1. Typical Techniques: Principles of Modularity and Use of Structured Code

Application of structured coding is widely accepted for the coding phase of the software life cycle. Structured coding refers to a methodology for problem solving as well as to the particular programming constructs used in code development. The structured coding methodology is a top-down approach to the application of the principle of modularity, i.e., that a program procedure should have only one function. Function in this context means the
transformation of input into output. A large problem is broken down into smaller subproblems. This process is repeated until the solution for the smallest subproblem is expressed as a single procedure.

Structured code requires the procedure to be written with single-entry single-exit programming constructs. The smallest set (D-structures) of constructs include:

1. the statement sequence,
2. the if-then-else for decision, and
3. the while-do for iteration.

It is well known that any program can be written with only these constructs. It is also accepted that structured coding is necessary to produce well-structured and easy-to-read-and-understand programs.

4.4.2.2. The DBSIM Techniques: Structured Coding Methodology, and Program Modification Control

Application of structured coding is widely accepted for the coding phase of the software life cycle. This coding technique is used in the MDBS implementation effort. We note that the practice of structured coding begins where the practice of the systems specification ends.
We also need to control modifications made to the files containing the program code. This is needed to avoid the situation where two programmers change the same file concurrently and they write the file; the second write will overwrite the changes made by the first programmer. The program modification control is achieved by having a list of files that are currently being modified. When a programmer wants to modify a file, he is asked first to check this list. If the file name is not in the list, he adds the file name to the list indicating that he is currently working on the file, and he then starts modifying the file. If the file name is in the list, indicating that another programmer is currently working on the file, he waits until the other programmer is done. When a programmer is finished modifying a file, he removes the file name from the list.

4.4.2.3. A Critique of the DBSIM Coding Techniques

Although we have adopted the tenets of structured coding for this project, we have neglected to provide precise standards for coding style. We have found that when a programmer has been assigned to continue coding begun by another programmer, he has edited the source files to fit his coding style. Precise standards for coding style might
have eliminated (or at least limited) the amount of editing. In particular, although all programmers have indented their programs, we have not set any standards for indentation. Even though many indentation standards would be acceptable, uniform indentation standards would contribute to readability of the listings.

We have not adopted conventions for naming variables and subroutines. As a result, when we have combined code written by different programmers, we have run into the problem of multiply-defined names. A naming standard and a dictionary would have helped to alleviate problems with multiply-defined function/variable names. A simple naming standard, which also results in producing descriptive names, is to have a prefix (or postfix) with each name that reflects the module it is part of. For example, let us say that there is a "request table module." Then, all procedures in this module can have the prefix RTM. This will reduce the problem of multiply-defined names when code written by different programmers are combined.

The technique for program modification control has not been part of the original techniques in DBSIM. We have realized the need for it after we have run into "update
problem" a few times. The lack of program modification control has not, however, been a problem during the coding phase of the implementation since the programmers have always been working on completely independent tasks. The problem has occurred when some students were carrying out performance analysis on version E of MDBS; the students had to add the code for collecting statistics to the programs. Different types of statistics were to be collected. As a result, there were times when two students were modifying the same file, each student adding the code for collecting a different type of statistics.

4.4.3. Validation

During the validation phase of the software life cycle, it is ensured that the developed system functions as originally intended. That is, it is validated that the system actually performs what it is supposed to do.

4.4.3.1. Typical Techniques: Verification and Testing

There are two approaches to validation: verification (or proof of correctness) and testing. Verification attempts to prove correctness of a program with respect to its specification. This is done by proving the validity of
certain assertions about the program outputs.

Program verification is still largely in the research stage. It is still almost impossible to apply it to large programs. So, testing is widely used for validation. There are three testing strategies: static, symbolic and dynamic.

In static testing, the software is analyzed (for determining errors) without actually executing it. For example, a compiler can be used to determine syntax errors in a program.

In symbolic testing, the names of input variables (as opposed to some values) are used to represent the inputs. The symbolic inputs are used and the code is "executed" to determine the symbolic values for output variables. The symbolic values for output variables are then analyzed to determine the presence of errors in the program.

In dynamic testing, sets of input values are selected and the software is actually executed on these input values. The output values are then analyzed to determine the presence of errors in the program. This strategy is the most widely used testing strategy.
4.4.3.2. The DBSIM Technique: Black-box Testing

In the **black-box approach** to testing, test data is selected without reference to the internal structure of the program. Instead, test data is generated based on the program functions described in the requirements analysis study. This approach is in contrast to the **structural approach** to testing, where test data is selected based on some characteristics of the internal program structure, for example, the number of paths through the program.

Intuitively, the black-box testing approach is applicable to testing database systems, since database users generally know more about the content of their databases than about the inner workings of the database system. Test data selected using the black-box approach will more closely resemble a realistic test of the system. Another advantage of the black-box approach is that, since no knowledge of internal program structures is required to develop the test data, it is easier to integrate into the testing phase the people who are not involved in the development phase.

One application of the black-box approach is functional testing [Howd80]. In this application, programs are viewed
as functions which map values from the program's domain of input variables into its domain of output variables. Test data is selected based on the important properties of elements in these domains. The functional testing method is particularly suited to the MDBS implementation. The requirements analysis study in [Hsia81a, Hsia81b] describes the functional components of MDBS and their input and output domains. For example, the input domain of directory management includes the set of retrieval requests. The output domain is the set of addresses of the records that may satisfy the requests.

4.4.3.3. A Critique of the DBSIM Validation Technique

There has been no formal application of the testing strategy. Some testing has been done by the programmers, and follows the familiar structural testing pattern. We have done most of the testing with a view of the entire system as a "black box." Data for such tests are developed based on the four request types of MDBS: insert, retrieve, delete, and update.

We have applied the concept of "unit testing", "integration testing" and "system testing" to some extent, but not completely. During the unit testing, individual
components, such as different procedures, are tested [Pres82]. During the integration testing, related components, such as all the procedures in a process, are combined and tested together. During the system testing, the whole system is tested. Unit testing is, obviously, more manageable than integration testing, and similarly, integration testing is more manageable than system testing. Furthermore, unit testing makes integration testing easier, and similarly, integration testing makes system testing easier. Application of unit testing, integration testing and system testing is a more practical approach to testing large systems and we should have followed it completely (we have tested some procedures individually which is unit testing; we have also tested some processes individually which is integration testing). Fortunately, we have not run into major problems. This can be attributed to the use of walkthroughs for the design and code; most errors have been detected before testing has started.

One problem that we have had during testing is related to array references. The C compiler does not generate code to check array references. So, out-of-bound references are not detected at run time. We have not originally realized that we should include code to check for exceeding array
sizes either. As a result, we have had to spend considerable time to find some of the errors in the programs.

It is estimated that 45-50% of the software effort is spent on testing [Boeh73]. Our software effort for testing has been noticeably less than this figure (the development chronology for MDBS is given in Chapter 6). The reasons for this departure from estimated testing time are:

(1) We have spent considerable time on the design.

(2) We have used walkthroughs for the design and code.

4.5. Concluding Remarks on the System Hardware and Software

Equipment delivery schedules have been such that the VAX is not delivered until May of 1982. For the first ten months, therefore, the development environment has been RSX-11M, with an unsupported C compiler. Also, the VAX is originally delivered with only 0.5 megabytes of primary memory. Until we have added memory in September, 1982, the response time has been very poor. We still feel that the choice of UNIX as a development environment has been a good one; however, we wish that it had been available much
earlier.

The run-time operating systems (VMS and RSX-11M) provide system features which are required for implementation of MDBS. Therefore, we have used facilities provided by these operating systems, instead of writing our own. This has reduced the implementation effort. We should have, however, chosen RSX-11M-PLUS instead of RSX-11M to reduce the problems with overlaying a process.

The systems programming language, C, is simple enough for the implementors to learn and powerful enough to program the functionalities of MDBS. Our primary problems in using C have been uninitialized variable references and out-of-bound array references. These are not detected by the C compiler and run-time support system. It would be helpful to have a compiler which includes these features for program development. Such features would greatly reduce program testing effort.

The concept of data abstraction used during the design phase of the software development has been very effective. It has helped us to produce well-structured programs. It has also made it easier to move from one MDBS version to another. The use of a systems specification language has
been helpful in communication among members. It has also reduced dependence on individuals since it provides complete and accurate documentation on the design.

The concept of structured programming used during the coding phase has helped us to produce well-structured, easy-to-read and easy-to-understand programs. We have not set standards for coding style (such as indentation style). This has resulted in waste of manpower due to reformatting of source files.

Our testing effort has been less than typical estimates such as that in [Boeh73]. This can be attributed to the use of walkthroughs for the design and code; most errors have been detected before testing has started.
5. SOLUTIONS TO THE ISSUES OF DATABASE SYSTEM IMPLEMENTATION

The issues of database system implementation have been introduced in Section 2.3. Unlike the management and system hardware and software issues which apply to all database systems, the database system implementation issues may vary from database system to database system. The issues in this category are: the interface with operating systems, the message-passing mechanisms, hardware and software requirements, the level of I/O operations, the organization of directory data and user data, the concurrency control mechanisms, and the supporting software. In this chapter, we describe special considerations in selecting solutions to these issues. The solutions adopted in the MDBS implementation and the rationale behind the choice of solutions are also discussed.
5.1. Two Approaches to the Interface with the Operating System

As described in Section 2.3.3, there are two approaches to interfacing with a run-time operating system: message-oriented and procedure-oriented. The two approaches differ in the process structure and process synchronization. Using a message-oriented approach, there would be a fixed number of processes, and synchronization is achieved by passing messages among processes. Using a procedure-oriented approach, there would be a varying number of processes, and synchronization is achieved by direct sharing and locking of common data in the main memory.

It is argued in [Laue79] that these approaches are equivalent. That is, we could expect systems implemented using these two approaches to be similar in functionality. Thus, in choosing one of these approaches to interface with an operating system, one must consider the efficiency that the operating system provides for each approach, and ease of structuring the database system. For example, the message-oriented approach requires exchange of messages between processes. If the underlying operating system does not have efficient message-passing facilities, the choice of a
message-oriented approach is not, obviously, appropriate.

5.1.1. The Choice of an Approach

The functional composition of MDBS allows either approach, message-oriented or procedure-oriented, to be used for implementing MDBS. However, two major problems may occur when using the procedure-oriented approach [Ston81]:

(1) Process switch overhead - When a process must be put to wait, a process switch is necessary in order to run another process. Process switching is costly because the information related to the blocked process must be saved and the processor scheduler must conduct considerable work to choose the next process to run. Since the procedure-oriented approach causes more process switches than the message-oriented approach, the process switch overhead is higher in this approach.

(2) Critical sections - Some processes have critical sections in which holds on locks are placed. If the processor scheduler deschedules a process while it is in its critical section holding some locks over some resources, all other processes will be queued up behind the locked resource. Thus, the database system performance will be degraded.

The operating systems, VMS and RSX, being used in MDBS facilitate message passing (we describe the message-passing facilities of VMS and RSX in Section 5.2). They also allow a process to receive messages from multiple processes. Because of the aforementioned two problems with the
procedure-oriented approach and because of the environment provided by VMS and RSX, we have decided to use the message-oriented approach.

The choice of the message-oriented approach also fits well with the architecture of MDBS. The decomposition of the functionality of each computer into processes results in defining different categories of functions and different categories of messages. This decomposition gives a well-structured design and it increases ease of understanding of the database system. The need for inter-computer communication in MDBS also fits well with the message-oriented approach since the entire system can now be viewed in terms of messages.

5.1.2. The MDBS Process Structure

Since a message-oriented approach is being used, we must choose a process structure for the controller and backends. There are several obvious choices. First, all the functions of a computer can be combined into one process. Since there is only one process in a computer, there is no concurrent execution of processes, i.e., a wait by the process will leave the CPU idle (there is no other process to execute). This alternative is, therefore,
unattractive because it will not utilize the processing unit efficiently. A greater degree of concurrency can be obtained by using multiple processes and the multiprogramming facilities of the underlying operating systems. An extreme alternative is to create a process for every basic function of the MDBS. While this does allow a high degree of concurrency, it is unattractive because of the great amount of message-passing overhead.

A third alternative is to use a smaller number of processes to facilitate concurrency, while keeping the message-passing overhead at an acceptable level. A good candidate organization is one which parallels the categories of functions which we have described in Chapter 3. There are three processes in the controller: request preparation, insert information generation and post processing. There are three processes in each backend: directory management, record processing and concurrency control. We have also chosen to implement the communication interface in each computer as two processes: get pol and put pol (we will discuss the rationale behind this choice in Section 5.2). Thus, the organization we have adopted for the process structure of MDBS consists of five processes in the controller and five processes in each backend.
5.2. The Inter-Process and Inter-Computer Message-Passing Mechanisms

Since a message-oriented approach has been selected for the MDBS implementation, there is a fixed number of processes in the controller and backends. The processes in a computer exchange information by sending and receiving messages. Thus, an inter-process communication mechanism is needed for the controller and for the backends.

The different computers also need to exchange information. For example, a request is broadcasted by the MDBS controller to the backends. As another example, each backend returns the results for a request to the controller, to be forwarded to the host machine. The MDBS design requires a broadcast bus for connecting the controller and backends. But, as we have described before, a replacement hardware must be used in the implementation. In the following sections, we describe the implementation of the message-passing mechanisms required in MDBS.

5.2.1. Message-Passing Within the Controller

The MDBS controller computer is a VAX-11/780 running the VMS operating system. For VMS, the inter-process
communication facility is a virtual device known as the mailbox [DEC80]. A mailbox is a software-implemented I/O device which can perform read and write operations. Let us examine how process A sends a message to process B. First, process A sends a message to process B's mailbox. Having issued a read on its mailbox, process B is given the message sent by process A. Thus the mailbox acts as a receiving queue. Messages remain in the mailbox until they are accepted by the receiving process.

We have implemented a service abstraction using mailboxes and made it available to the processes in the controller. This abstraction provides procedures for sending and receiving messages. Thus, this abstraction relieves the processes in the controller from performing such functions as waiting for a message when there is not one and indicating that there is a message when one arrives. The "send" and "receive" procedures provided by this abstraction are called whenever a process needs to send or receive a message. The use of this service abstraction also makes the processes in the controller independent of the underlying message-passing mechanisms, since the facility (mailbox in this case) used by the "send" and "receive" procedures can be replaced without affecting the calling
processes.

5.2.2. Message Passing Within a Backend

As stated earlier, the backends are PDP-11/44s running the RSX-11M operating system. Under RSX-11M the inter-process communication facility available to us is the shared access to the physical memory [DEC79a]. Suppose process A wants to send a message to process B. First, A copies the message into a shared area of the memory. Then A asks the operating system to send a pointer to this area to process B. The operating system maintains a queue of pointers for process B. When B is ready to use a message, it gets the pointer from the operating system and it then copies the message into its own memory.

The RSX operating system also provides facilities for sending/receiving a short message (13 words) directly. Though this facility is easier to use than the shared access to physical memory, it is not satisfactory for the MDBS implementation because MDBS has long messages.

As in the controller, we have employed the concept of service abstraction in implementing the message-passing mechanism in the backends. That is, we have implemented a
service abstraction (using the shared access to physical memory) and made it available to the processes in the backends. This abstraction provides procedures for sending and receiving messages. The "send" and "receive" procedures provided by this abstraction are called whenever a process needs to send or receive a message.

5.2.3. Message Passing Between Computers

The hardware for connecting the computers provides the mechanism for one machine sending a message to another machine. The broadcast capability is not available in the hardware. Therefore, we have simulated the necessary broadcast feature in MDBS. Communication between computers in MDBS is achieved by using a time-division-multiplexed bus called parallel communication link (PCL-11B or PCL for short) [DEC79b]. The PCL uses a polling sequence to give each computer a time slice to send one 16-bit word over the bus. A message of n words must be sent over the bus sequentially as n messages, each message of one word in length. The PCL driver (provided by DEC) provides the mechanism for sending/receiving a message of n words, i.e., the PCL driver breaks a message of n words into n messages of one word each and sends them over the bus; it also
receives \( n \) messages of one word each and puts them together as a message of \( n \) words.

There can be only one process in each computer which gets messages from the PCL. This process needs to declare to the PCL that it is the sole receiver of messages coming to the resident computer over the PCL. For this reason, the communication interface in each computer includes a process, \texttt{get pcl}, which gets messages from other computers off the PCL. The \texttt{get pcl} process is responsible for receiving messages sent to its computer and for forwarding them to the specified MDBS processes in the same computer.

Any process in a computer can transmit a message over the PCL to another computer. We have, however, chosen to have a second process, \texttt{put pcl}, in each computer for sending messages to other computers. The reason for this choice is that the transmission of messages is logically a different task. This separation results in a well-structured design. It also reduces the address space of other processes since the interface code (required for sending messages over the PCL) is not part of those processes. The \texttt{put pcl} process could also provide monitoring, scheduling and blocking of messages. Together with \texttt{get pcl}, \texttt{put pcl} forms a service
abstraction that provides mechanisms for communication between different computers.

5.3. Simulation of the Broadcasting Hardware

The MDBS design requires a broadcasting bus for connecting the controller and backends (see Figure 2). Since a broadcasting bus is not originally available, the MDBS implementors have used a parallel communication link (PCL) for this purpose. The PCL allows transfer of a message from one computer to another. It does not have the broadcasting capability. As a result, we must simulate the broadcasting feature. For example, let us say there are two backends in MDBS and the controller needs to broadcast a message. This is done by sending two identical messages, one to each backend.

The put pcl process of the communication interface transmits messages over the PCL to some other computer's get pcl process. If the put pcl process is in the controller, then it will transmit every message to each backend (i.e., broadcast). In a backend, there are two possibilities. The first is that a message is to be sent to the controller. This means a single transmission. Secondly, a message is to
be broadcasted to other backends, then in the present simulated mode it will be sequentially transmitted to all the other backends. If a transmission by a put pci process fails to be accepted by the destination computer, then the sending process will wait and try the transmission later. For example, if computer A sends a message to computer B, while computer C is sending a message to computer B, then computer A's message will not be acknowledged by the get pci process in computer B. Therefore, the put pci process in computer A will wait a certain amount of time and retransmit the message. (These mechanisms, simulating the broadcast and trying the message transmission until it is acknowledged, are provided by the communication-interface service abstraction. That is, other MDBS processes are not concerned with these details.)

This replacement hardware is acceptable when the number of backends in MDBS is small (say, 2 or 3). The replacement hardware will become a bottleneck as the number of backends in MDBS increases. For example, if there are 10 backends in the system, broadcasting a message from the controller to the backends requires the transfer of 10 messages on the PCL! The current research effort of MDBS implementors includes investigation of local-area networks in order to
find an appropriate replacement for the PCL. In particular, we are investigating different versions of Ethernet [Metc76]. Our plan is to use this type of local-area networks in the next prototype of MDBS.

5.4. The Input/Output Operations

Two issues must be addressed for I/O operations in a database system. First, the size of unit to be stored onto and retrieved from the disk must be chosen. The goal in this selection should be to minimize the number of I/O operations and, at the same time, reduce the unused space on the disk.

The second issue that must be addressed is the selection of the level of I/O operations. Different levels of I/O operations are usually provided by the operating system. The goal in choosing an I/O level in a database system implementation should be to minimize the implementors' work and, at the same time, meet the system requirements. We discuss these two issues in the following sections.
5.4.1. The Size of Unit for I/O Operations

The MDBS design requires that the records in a cluster be stored in close proximity, since any access to the disk for a non-insert request is based on one cluster at a time. Thus, the best unit size for I/O operations is one that minimizes the number of I/O operations required to access all the records in a given cluster and, at the same time, reduces unused space on the disk. The analysis in [Hsia81a, Hsia81b] indicates that the optimal choice is to group records in a cluster into tracks, to divide the tracks of the cluster evenly by the number of backends, and to read (write) the quotient tracks of data from (to) the disks of the respective backends simultaneously. A smaller unit for grouping the records in a cluster will increase the number of I/O operations required to access the records in a cluster. A larger unit will increase the unused space on the disk.

The disk drives we are using on our PDP-11/44s perform actual input and output on 512-byte sectors, i.e., when issuing an I/O operation, the user has to specify the number of sectors to be read/written. (Each disk pack, RM02, has 823 cylinders, 5 tracks per cylinder, 32 sectors per tracks,
a total of 131680 sectors, and a capacity of 67 megabytes. The average seek time is 30 ms; the average latency time is 12.5 ms; and thus the average access time is 42.5 ms. The peak transfer speed is 806 kilobytes/second.) Let us refer to the unit for I/O operations in the MDBS implementation as a block. Then, the decision to be made in our implementation is how many sectors there should be in a block, i.e., should a block be one sector, two sectors, or more sectors? As described above, the analysis in [Hsia81a, Hsia81b] indicates that the optimal choice is to group records in a cluster into tracks. So, the optimal choice is to have 32 sectors in a block, to divide the blocks of a cluster evenly by the number of backends, and to read (write) the quotient blocks of data from (to) the disks of the respective backends simultaneously. We have, however, chosen one sector as the unit for I/O operations in the current MDBS implementation, i.e., a block has only one sector (and not the optimal 32 sectors). Even though a larger block size improves the system performance, it requires extra memory space for I/O buffers. Since both virtual and physical memory are a constraint in the current hardware configuration, we have not chosen a larger block size, say a track, in order to alleviate the problems with
virtual and physical memory. (The MDBS implementation does, however, allow that the block size to be increased easily since the block size is a parameter to I/O operations. That is, if virtual and physical memory are not a constraint in the next hardware configuration, we can perform I/O operations in terms of larger block size without any change to the current code.)

5.4.2. The Level of I/O Operations

Many alternatives are available to us to achieve this type of I/O operation. We can use a very low-level interface with the disk device driver software. We can also use a very high-level file system interface. Our goal is to minimize our work while still obtaining good system performance. We have decided that neither of these approaches is appropriate. One is at too low a level that requires unnecessary work by the implementors; the other is at too high a level that degrades the system performance. Thus, we have looked for a level in between. Because of our lack of experience with the RSX operating system and because of poor documentation of what exactly is available, this interface has been very hard for us to develop.
Below the file level, RSX provides three types of input/output support. These are called virtual, logical and physical I/O. The physical I/O interface requires the user to specify the actual physical address on the disk. In this case, the user must also keep track of bad sectors on the disk. At the logical I/O interface, the system automatically skips bad disk sectors. Thus, the user does not have to manage these bad physical sectors. At the virtual I/O level, the user is allowed to use arbitrary record size rather than being restricted to 512-byte sectors.

Our major difficulty has been in determining what these actual interfaces are since most documentation only discusses the file system interface. Once we have understood these three interfaces, it has not been hard to make the choice. The additional time required to manage arbitrary record size of the virtual I/O makes this alternative less efficient. We have rejected the virtual I/O level because we are actually trying to read or write a whole sector of data (which is, of course, of fixed size) and because we want to make our system as efficient as possible. We have decided to use the logical rather than physical level since we would then not be required to
develop software to manage bad sectors on the disks, thus avoiding unnecessary work by the implementors. Since the purpose of our implementation of MDBS is to show its feasibility, our experiments will use only disks without bad sectors of data. MDBS performance would be as good as if we had handled the bad sectors ourselves, i.e., using the physical I/O. In this way, we are relieved from having to handle bad sectors ourselves, and have achieved the necessary performance.

5.5. Implementations of Directory-Data and User-Date Structures

As described before, database systems generally have two types of data: directory data and user data. The database system implementors need to investigate different organizations to represent and store these two types of data. Appropriate organizations must be selected so that the system performance is satisfactory.

The project planners may decide that more than one implementation to represent the directory data and/or the user data should be carried out. They may decide that the first implementation should be a simpler one as an exercise
to gain experience (a simplified implementation also allows
testing of the rest of the system in a less complex system).
The project planners may also decide that the first
implementation should be a satisfactory one so that its
performance can be compared with alternative implementations
to choose the best one. For example, the directory data may
be implemented twice: using, say, hashing first and B-trees
next. The performance of these two implementations can then
be compared to choose the best one.

In MDBS, directory data is maintained and used for fast
and efficient retrieval of user data. We have implemented
two directory-data structures and two user-data structures.
The first implementations are simple but not acceptable in
real applications. The second implementations are
complicated but required for real applications. The
implementations of simple versions have prepared us for a
better second implementation. This application of the
build-it-twice-different-versions strategy has also allowed
us to develop and test other functionalities of MDBS in a
simpler prototype.
5.5.1. The Directory-Data Structures

The first implementation of directory data uses the primary memory for storing the directory information, so the directory management for this implementation is referred to as the primary-memory-based directory management. Arrays and linked lists are used in this implementation. As computer science students, we are familiar with these data structures. Thus, the implementation of this simple directory management has been a good exercise in systems programming for us. This implementation is not, however, feasible for supporting large databases since the directory data would be too large to be kept in the primary memory. The primary-memory-based directory management is used in versions A through E of MDBS.

The second implementation of directory data uses the secondary memory for storing the directory information, so the directory management for this implementation is referred to as the secondary-memory-based directory management. B-tree and B⁺-tree organizations are primarily used in this implementation. We have studied several different organizations and have investigated several alternatives. Our goal is to implement an efficient directory management
since the performance of the directory management is a major factor in the performance of the system (MDBS). The secondary-memory-based directory management is used in version F of MDBS.

5.5.2. The User-Data Structures

The implementations of the user-data structures are also based on the primary and secondary memory. We have used a data abstraction to implement the disk subsystem. The first implementation of the disk subsystem uses the primary memory for storing the user data. A multi-dimensional array is used to simulate a disk in this implementation. We have programmed two routines, "fetch" and "store," to perform the read and write operations on disk. The higher-level routines use these two routines for operations on disk. This application of the data abstraction concept is intended to simplify the replacement of the simulated disk. This implementation of a disk has allowed us to develop and test other functionalities of MDBS under UNIX which provides a better development environment than RSX (final version of MDBS uses the RSX as the run-time operating system in the backends). The simulated disk is used in versions A, B and C of MDBS.
The second implementation of the database storage uses the secondary memory. For this version, we have replaced the code for "fetch" and "store" routines to perform actual I/O (see Section 5.4 for a discussion of the I/O). This change has not affected the higher-level routines, so no modification at higher levels have been required. The secondary storage I/O is used in versions D, E and F of MDBS.

5.6. The Concurrency Control for User Data and Directory Data

Concurrency control is needed in a database system to allow concurrent and interleaved execution of requests. Access to the directory data as well as access to the user data must be controlled in order to provide correct results (responses) to the database system users. The mechanism for concurrency control in a database system must be studied thoroughly so that its correctness and completeness are verified. The complexity of the concept of concurrency makes this task very difficult.

In MDBS, concurrent and interleaved execution of requests is allowed in order to maximize system throughput.
Thus, concurrency control mechanisms are needed to preserve the integrity and consistency of the database, and to provide correct results (responses) to the database system users.

The directory data and user data are kept in the MDBS backends. Thus, a concurrency control mechanism is not needed in the MDBS controller. In the following sections, we describe the concurrency control mechanisms in the backends.

5.6.1. The User-Data Concurrency Control

In MDBS, insert, delete and update requests change the contents of the database. The user-data concurrency control mechanism in MDBS locks the clusters needed by a request before the request can operate on the records in the clusters. After the request is finished, the user-data concurrency control mechanism releases the locks on the clusters. If a request can not lock a cluster due to a lock placed on the cluster by another request, the request is placed in a wait queue.
5.6.2. The Directory-Data Concurrency Control

In this section, we demonstrate that concurrency control is needed for directory data and then we describe the concurrency control mechanism for directory data. Since the purpose of this report is to describe the implementation issues and the database systems engineering experience rather than the database system itself, the descriptions presented in this section are brief, simplified and informal. Details on the concepts and terminologies used in MDBS can be found in [Hsia81a, Hsia81b, He83, Kerr83, Boyn83a, Boyn83b, Demu84].

5.6.2.1. The Directory Information

Before describing the implementation issues for directory information, it is necessary to give a brief description of the directory tables. The data model used in MDBS is the attribute-based data model [Hsia70]. In MDBS the database consists of files of records. For performance reasons, records are grouped into clusters based on the attribute values and attribute value ranges in the records. These values and value ranges are called descriptors. For example, one cluster might contain records for those employees making at least $20,001 but not more than $25,000
and with rank 2. Thus records of this cluster are grouped by the following two descriptors:

\[(20001=\text{SALARY}<25000), (RANK=2)\]

Record retrieval in MDBS, for example, is done by clusters. Thus finding records of employees making $21,000 and with rank equal to 2 would require the retrieval of records in the cluster just described. Other requests such as to find records of employees making between $21,000 and $28,000 and with rank equal to 2 might require additional retrieval of records from clusters other than the one identified above.

Let us now describe how descriptors are defined and clusters are formed in MDBS. The database creator specifies some attributes of the database as **directory attributes**. The directory attributes are stored in a table called the **attribute table** (AT). For a directory attribute, the database creator can either specify some descriptors or decide that every value for the attribute should be a separate descriptor. In the latter case, the system generates a new descriptor whenever a new value is encountered for the attribute at the record-insertion time. The descriptors are stored in a table called the **descriptor-to-descriptor-id table** (DDIT). Those attribute values in a record whose attributes appear as the directory
attributes specify a set of descriptors. This descriptor set determines the cluster to which the record belongs. Different sets of descriptors determine different clusters. The cluster definitions (in terms of the descriptor sets) and secondary storage addresses of the records in the clusters are stored in a table called the cluster-definition table (CDT). AT, DDIT and CDT contain the directory information in MDBS.

5.6.2.2. The Need for Concurrency Control

DDIT contains the descriptors defined in the system. Some descriptors are defined and stored in DDIT at the database-load time. Some descriptors are, however, generated and stored in DDIT as new records with new values for certain directory attributes are being inserted into the database. So, DDIT, a directory table, may be updated in real-time as a result of user insert requests.

CDT contains the cluster definitions and secondary storage addresses of the records in clusters. New clusters are generated whenever there is a new record whose corresponding descriptor set is different from all the existing descriptor sets. So, CDT, another directory table, may also be updated as a result of user requests.
Since directory data may be updated as a result of user requests, concurrency control is needed for the directory data if concurrent and interleaved execution of requests is to be allowed. In the following section we describe the concurrency control mechanism used in MDBS to insure the consistency of the directory data.

5.6.2.3. The Mechanism for Concurrency Control

Some relational database systems may store directory data and user data the same way, i.e., relations may be used to represent both types of data. Thus, in these relational systems, only one concurrency control mechanism is required, although the mechanism may be called upon several times. In MDBS, however, the structures for directory data and user data are different. As a result, two different concurrency control mechanisms are required: one for directory data and one for user data.

As described in the previous section, DDIT and CDT may be updated as a result of user requests. Thus, access to these tables must be controlled for concurrent and interleaved execution of requests. The directory-data concurrency control mechanism in MDBS locks parts of a table (DDIT or CDT) that are needed by a request before the table
is accessed. The locks on a table (DDIT or CDT) are released after the request has been processed and the table is no longer needed for the request. The directory attributes needed by a request are used to determine which parts of DDIT must be locked before accessing the table, since descriptors (in DDIT) are defined on directory attributes. The descriptors needed by a request are used to determine which parts of CDT must be locked before accessing the table, since clusters (in CDT) are defined in terms of descriptors.

5.6.3. Our Experience with Concurrency Control

As mentioned before, designing the concurrency control mechanisms for a database system is one of the most, if not the most, challenging tasks in database system design. The complexity of the concept of concurrency makes this task difficult. As we have explained in Chapter 3, our overlook in this area has resulted in modifying the code, which has already been completed, for the processes in the MDBS backends. This has resulted in loss of effort.

We have not addressed the issue of concurrency control implementation until version E of MDBS, i.e., we have not investigated the impacts of concurrent request execution
until version E (the MDBS development strategy is described in Chapter 3). Because of the complexity inherent in concurrency control and because of its impacts on other functions of a database system, the concurrency control mechanism should be considered from the early stages of an implementation effort. We can state from our experience that other functionalities of a database system, compared to concurrency control, are easy to comprehend and the problems in them are detected easily. The concurrency control mechanism, on the other hand, is difficult to understand and the problems in it are hard to detect. For careful and considerable investigation of the concurrency control mechanism, we have taken extra time.

5.7. The Additional Software for Experimenting with the System

One of the goals of our implementation effort has been to validate the simulation results on MDBS described in [Hsia8la, Hsia8lb]. Consequently, as a preparation for this experiment we have developed supporting software consisting of three programs: a "file generation" program, a "database load" program and a "user test interface" program. The
file-generation and database-load programs have been implemented before beginning the system implementation effort itself. The early implementation of the supporting software has prepared us (inexperienced implementors) for MDBS development. A simplified version of the user-test-interface program has been implemented during early MDBS development effort. It has been completed before version A of MDBS has been developed. We have augmented this program with other desirable features during MDBS implementation.

The file-generation program generates test data files. The values for an attribute in a test data file are generated randomly. A range of values or a set of values is provided by the user as the domain for each attribute. Program-generated test data has been used for two purposes. First, we have used this type of data to test each version of MDBS to see that it works correctly. Second, as described before, the initial performance evaluation experiments uses program-generated data.

The database-load program handles the initial loading of a database from an existing file. This program allows a user to define directory attributes and descriptors. After this directory information has been defined, the program
loads the database and creates the directory tables (AT, DDIT and CDT).

The user-test-interface program allows a user to create and modify files of test requests. These requests can then be submitted to MDBS by giving a single command. This interface makes it easy to run demonstrations and tests of MDBS. As mentioned before, we have added new features to the test-interface program as we needed. This has worked better than specifying the complete requirements at the beginning, which is the case for program development. The reason for this is that the "human factor" has a major impact on programs such as the test-interface program. As a result, it is not easy to specify the requirements for such programs at the beginning. Instead, these programs should be used by the users and they can easily specify new desirable features. These features can then be augmented to the original programs.


Even though the database system implementors are given the requirements specification of the system by the
originator of the system, several decisions must be made by the implementors. The management issues and system hardware and software issues have been discussed in Chapters 3 and 4. In this chapter, we have discussed issues specific to database system implementation in general and to the MDBS implementation in particular.

The primary concerns in selecting solutions to these issues are to minimize the implementors' work and, at the same time, meet the system requirements. First, an approach (either procedure-oriented or message-oriented) for interfacing with operating systems is selected. The facilities provided by the underlying operating systems for each approach and ease of understanding of the database system must be considered in this selection. The message-oriented approach selected for the MDBS implementation has resulted in the use of facilities provided by the underlying operating systems. This has reduced the implementors' effort. The message-oriented approach has also resulted in a well-structured design. This has increased ease of understanding of the system.

The inter-process and inter-computer communications have been implemented using the facilities provided by the
RSX and VMS operating systems and the PCL hardware. We have used the concept of data and service abstractions in implementing the required message-passing mechanisms. This has resulted in a well-structured design. It has also made the MDBS processes independent of these mechanisms, i.e., the message-passing mechanisms can be replaced by other implementations without affecting the MDBS processes.

The broadcasting capability required in MDBS must be simulated. The PCL is acceptable when the number of backends in MDBS is small (say, 2 or 3). The PCL will become a bottleneck as the number of backends in the system increases. Investigations on local-area networks, such as Ethernet, as a replacement for the PCL are underway.

The implementation of the I/O operations has taken a long time. The reason for this is our lack of knowledge in operating systems. Most system manuals (such as the operating system manuals) are also written for experienced people, which has made our task more difficult. The choice of a unit size for I/O operations has been affected by the limitation on virtual and physical memory. More specifically, we have used a smaller unit size than the optimal unit size, in order to reduce the memory space.
required by the I/O buffers.

We have implemented two versions of directory-data and user-data organizations: one using the primary memory and the other using the secondary memory. The implementation of the simpler, primary-memory version has helped us to gain some experience. It has also made it possible to test other functionalities of MDBS in a simpler system. The use of data abstractions in implementing the directory tables and the disk I/O subsystem has made it easy to replace the primary-memory-based implementations by the secondary-memory-based implementations. More specifically, no changes to the higher-level routines (the calling routines) have been required when going from one version to another.

The concurrency control mechanism has been one of the difficult parts for us to comprehend and implement. This difficulty is mainly attributed to the complexity inherent in concurrency control. It is also because we have not addressed this issue until the latter stages of the development effort. We have not investigated the impacts of concurrent request execution until version E of MDBS. Because of the impacts of concurrency control on other functions of a database system, the concurrency control
mechanisms should be considered from the early stages of an implementation effort.

Finally, we have implemented some supporting software programs in the early stages of the development effort. The implementation of these programs has prepared us for the development of MDBS. They have also been useful for testing MDBS.
6. TOWARDS A COMPREHENSIVE METHODOLOGY FOR THE IMPLEMENTATION OF LARGE-SCALE DATABASE SYSTEMS

The implementation of MDBS has begun in the summer of 1981. The implementation of version F (last version to be implemented) has been completed in the summer of 1984. Now, at the end of the implementation project, is the time to assess our progress toward our goals. Have we developed a workable, comprehensive methodology for the design and development of database systems? Have we made a quantitative contribution to the engineering of database software? Have we made a qualitative contribution to software engineering? We feel that we have at least partially succeeded in all of these areas.

In this chapter, we first give some statistics about the system software that we have developed. We then give a summary of the methodology used in the MDBS implementation. Finally, we examine our implementation effort to summarize our experience.
6.1. Facts and Figures

We have collected some facts and figures about the software we have developed. We find, after the fact, that we have not collected all the statistics we would like to have. We have not accumulated man-hour figures for design, coding, and testing. We do, however, have figures on the size of the system, and information from our walkthrough reports which pinpoints the chronology of development.

6.1.1. The Code Volume

As described earlier, MDBS consists of three processes in the controller, three processes in each backend, and a communication interface at each machine. The communication interface, which actually transmits messages from one computer to another, has been implemented as two processes. The first, get pcl, receives messages, and the second, put pcl, sends messages.

6.1.1.1. The Controller

The C source programs for the MDBS controller, including the communication interface processes, consist of 2849 lines of C code, and 2039 lines of comments. The
sizes of object modules in kilobytes (KB) are as follows:

<table>
<thead>
<tr>
<th>Module</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request Preparation</td>
<td>35.3KB</td>
</tr>
<tr>
<td>Insert Information Generation</td>
<td>14.8KB</td>
</tr>
<tr>
<td>Post Processing</td>
<td>14.3KB</td>
</tr>
<tr>
<td>Get pol</td>
<td>10.7KB</td>
</tr>
<tr>
<td>Put pol</td>
<td>10.2KB</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>85.3KB</strong></td>
</tr>
</tbody>
</table>

The largest part of the controller is the request preparation process. The request preparation code includes the request parser.

6.1.1.2. The Backends

The C source programs for each backend consist of approximately 10125 lines of C code, and 9485 lines of comments. As described before, we have implemented a primary-memory-based and a secondary-memory-base directory management. The primary-memory-based directory management consists of approximately 2494 lines of C code, and 1947 lines of comments. The secondary-memory-based directory management consists of approximately 5727 lines of C code, and 5686 lines of comments. The C code common to these two versions of directory management is approximately 1601 lines. This is about 66% of the primary-memory-based directory management, i.e., the majority of the code for the
first version is used in the second version. This is attributed to the use of data abstractions in implementing the directory tables; the major change required when going from the first version to the second version is that of replacing the data abstractions with new ones.

The space required for object code in each backend is shown below:

| Secondary-memory-based Directory Management | 92.3KB |
| Record Processing                          | 42.2KB |
| Concurrency Control                        | 50.0KB |
| Get pci                                    | 23.8KB |
| Put pci                                    | 23.9KB |
| **Total**                                  | **232.2KB** |

The operating system (RSX-11M) requires 74 kilobytes. An additional 8 kilobytes per process is required for message-passing. Finally, each buffer for disk input and output requires 16 kilobytes. We have used an overlay structure to remain within the PDP-11/44 address-space limitation (64 kilobytes) and within the limitations of the physical memory size of the current configuration (256 kilobytes).

The above figures for the object code in the controller and backends do not include the space for debugging code. We estimate that, on the average, each process has 6
kilobytes of debugging code.

6.1.2. The Development Time

The MDBS implementation team has been organized during June of 1981. The development of Version F (last version to be implemented) has been completed in August of 1984. The MDBS implementation has taken, as is so often the case, twice as long as the first estimated. The reasons for this long development time can be summarized as follows:

(1) The implementors have not had any experience, especially in operating systems, when they start the implementation effort.

(2) The implementation team members have been continuously changing.

(3) We have not realized that concurrency control is also needed for the directory data until winter quarter of 1983. This late detection has caused us to modify the code which has already been completed. This has resulted in loss of effort.

(4) The C compiler does not generate code to check array references. So, out-of-bound references are not detected at run time. We have not originally realized that we should include code to check for exceeding array sizes either. As a result, we have had to spend considerable time to find some of the errors in our programs.

(5) Versions A through D of MDBS support multiple databases. We have not originally realized that MDBS should also be capable to support multiple files in each database. We have had to change most of the MDBS processes to include provision for multiple-file databases (this change has been made when we were developing Version E from
Version D).

The MDBS implementation team has met weekly since the original organization of team in June of 1981. The statistics that follow have been collected from the minutes of the weekly meetings and from the walkthrough reports.

The first six months of the effort have been spent learning to use the C language, becoming familiar with the available development tools, and learning how to apply our software engineering techniques. The initial development effort has consisted of three projects. Of those, two are support software. Only one, the primary-memory-based directory management, is to become an integral part of the early versions of MDBS.

Version A, the simplest of all versions, has not been completed until August, 1982, more than a year after the implementation effort has begun. Subsequent versions have, however, followed more quickly. Figure 14 summarizes the progress of the implementation through six versions of MDBS. In the sections which follow, we include more detail on the progress of the implementation.
<table>
<thead>
<tr>
<th>Version</th>
<th>Completion Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. Single Process, Single Minicomputer, Virtual Disk</td>
<td>August, 1982</td>
</tr>
<tr>
<td>B. Multi-process, Multi-computer, Single Backend</td>
<td>December, 1982</td>
</tr>
<tr>
<td>C. Multi-process, Multi-computer, Multiple Backends</td>
<td>January, 1983</td>
</tr>
<tr>
<td>D. Secondary Storage I/O Subsystem</td>
<td>February, 1983</td>
</tr>
<tr>
<td>E. Concurrency Control</td>
<td>December, 1983</td>
</tr>
<tr>
<td>F. Secondary-memory-based Directory Management</td>
<td>August, 1984</td>
</tr>
<tr>
<td>G. Access Control</td>
<td>(Not to be implemented)</td>
</tr>
</tbody>
</table>

Figure 14: MDBS Versions and Completion Dates
6.1.2.1. Version A

That this version has taken the largest portion of the development time is not surprising. In order to implement this single process on a single computer, we have had to implement most of the functionality of MDBS.

The first portion to be implemented is the primary-memory-based directory management function. Design has begun in July, 1981, and has been completed in September, 1981. The design work has been shared by two students. Two students have begun coding in September, 1981. The coding has been completed in December, 1981. Design for the record processing function has begun in January, 1982, and has been completed in May of the same year. The design has been the work of one student (a visiting scholar). Coding has started in May and has been completed in July, 1982, by a single student. The concurrency control function, which is the third component of the backend, is not required for this version.

All three controller functions, request preparation, insert information generation and post processing, have been designed during the period from January through July, 1982. The long design time is due in part to a switch in chief
designers midway through the design. A chief designer and one other student have done the work. Only the first two of the three functions of the controller have been implemented for Version A. One student has begun coding the parser in March, 1982, using LEX [Lesk79] and YACC [John78b] from the UNIX tool-bag. Two students have finished the coding for request preparation and insert information generation in August, 1982.

6.1.2.2. Version B

In order to implement this version, we need to implement both inter-process and inter-machine communications. Preliminary experiments with the parallel communication link (PCL) have been done by one student during the spring quarter of 1982. Another student has continued the work into the summer quarter. Implementation of the inter-machine communication processes has been completed in December, 1982. Our lack of systems experience has been a major factor contributing to the extended development time.

A faculty member and a student have implemented inter-process communication in RSX and VMS, respectively, in the autumn quarter of 1982. Version B, the multi-process,
multi-computer version with a single backend, has been completed and tested in December, 1982.

6.1.2.3. Version C

Implementing Version C requires only that we add a second backend. Since all backends run identical software, no code changes have been required. Some problems have been identified and corrected during testing. Testing has been completed in January, 1983.

6.1.2.4. Version D

In Version D, we have added the secondary storage I/O subsystem. At least one person has worked on this implementation task from January, 1982, through January, 1983. Again, our lack of systems programming experience has proved a significant handicap. The I/O subsystem has been installed and tested in February, 1983.

6.1.2.5. Version E

Versions B, C and D of MDBS all represent the implementation of operating system-dependent tasks. In Version E, we are once again concerned with increasing the functionality of MDBS. Understanding and implementing the
concurrency control algorithm of MDBS has been a non-trivial task. Design for user-data concurrency control has begun in April, 1982, and has not been completed until January of 1983. Two students have worked successively on the design. One student has coded the user-data concurrency control, beginning in January, 1983, and completing the code in February, 1983.

We have not realized that concurrency control is also needed for directory data until winter quarter of 1983. A student has completed the design for directory-data concurrency control in spring quarter of 1983. A student and a faculty have made the required changes in MDBS code in summer 1983. Testing has been completed in December, 1983.

6.1.2.6. Version F

This version, the last which we have implemented, has a secondary-memory-based directory management. (We do not plan to implement Version G. The system through Version F is sufficient for our research into performance evaluation.) We have devoted a lot of time to the design, since the performance of the directory management is a major factor in the performance of the system. A preliminary survey of the secondary-memory-based file organizations has been done.
during the autumn quarter of 1981. A second study has been conducted during the spring quarter of 1982. Detailed design has not begun until September, 1982. The design, primarily the work of one student, has been completed in December. One student has completed part of the coding in winter quarter of 1983. Because of the problems seen in concurrency control algorithm and because of concentration on Version E, nobody has continued the coding for Version F in spring and summer quarters of 1983. Two students have completed the coding for this version in winter quarter of 1984. Testing has been completed in August, 1984.

6.1.3. The Development Chronology

Figure 15 is a pictorial representation of the development chronology. For each version of MDBS, design and coding time spans are delineated, along with total development time. We notice some interesting items.

First, look at Version A. The first piece of the MDBS system to be implemented, the primary-memory-based directory management, has taken four months to code, and only three to design. In other cases, design time is longer than coding and testing time! This is an interesting departure from the
Figure 15: MDBS Development Chronology
figures in [Boeh73], which show 45-50% of the software effort in testing. Even more interesting is the fact that, as the implementation effort has progressed, design times have lengthened in proportion to coding times.

Another observation we can make is that for operating system-dependent tasks, no separate design and coding times are shown. (The operating system-dependent tasks in MDBS are: inter-process communication, secondary storage I/O subsystem and inter-computer communication.) That the software engineering methodology has not been applied in these cases may indicate that

(1) we have overlooked to employ software engineering techniques;

(2) software engineering techniques have not been widely applied to the development of such tasks, so people are not used to this concept yet; or

(3) software engineering techniques are not as suitable for such tasks as they are for other tasks.

We also note the relative lack of activity shown for the first seven months of the effort. This is partly explained in that the support software tasks are not included in the chart. It also reflects that the first months of the effort have been spent in learning activities. It is also interesting to note that in 1982, more than 40%
of the effort has been directed toward the operating system-dependent tasks.

The design and coding time in Figure 15, estimated at 20 person-hours per week, four weeks per month, represents 3 person-years of effort. We estimate that we could have accomplished the system-dependent tasks in about half the time with experienced programmers. This would reduce the time by approximately one-half person-year.

Despite the careful application of our software engineering methodology, the implementation has taken twice as long as the first estimated. We do feel that the first estimates, as is so often the case, have been naive. A more important issue is whether the end product is reliable. Our intent is to use the prototype to verify performance predictions and to conduct performance evaluation experiments. We will also collect data on software failures during the performance studies and evaluate the reliability of the MDBS software.

In the preceding sections we gave the statistics that we have collected during the implementation effort. In the following sections, we first give a summary of DBSIM. We then summarize our experience and present our observations.
6.2. A Review of DBSIM

The database systems implementation methodology (DBSIM) can be summarized as follows:

Management Techniques
  Incremental Development
  Chief-Programmer-Team Organization
  Formal Reviews for Design and Code

Software Development Techniques
  Design Techniques
    Top-down Design and Use of Data and Service Abstractions
    Systems Specification Language
    Use of Jackson Chart
  Coding Techniques
    Principles of Structured Coding
    Program Modification Control
  Validation Techniques
    Black-box Testing Approach

As software engineers, we believe that techniques for management and software development must be chosen for software effort. The management techniques should include a development strategy, a team organization and techniques for software quality assurance. The software development techniques should include techniques for different phases of the software life cycle.

Overall, we believe that the methodology we adopted has been effective and has worked well. A more comprehensive look at the implementation effort is given in the following
6.3. An Examination of the Present Implementation Effort for a more Effective Future Implementation Undertaking

After three years of effort, implementation of MDBS is finished. Now, at the end of implementation effort, is the time to take another look at the whole process and answer questions such as:

- What have we done right?
- What have not we done right?
- What would we do differently next time?
- What techniques have worked well?
- What techniques have not worked well?
- How can the techniques be improved?

Answers to such questions will be useful for future implementation efforts. In the following sections, we summarize the implementation effort and our experience.

6.3.1. The Development Environment: The Need for Expertise in Operating Systems, and Usefulness of Software Tools

We have selected UNIX as a development environment, and the C language as the systems programming language. The
software has been developed on the VAX system running UNIX. The actual prototype, however, runs under the VMS operating system on the VAX and RSX-11M on the PDP-11/44s.

Our lack of knowledge in operating systems has been a major problem (by lack of knowledge in operating system we mean lack of systems programming experience using operating system facilities such as low-level I/O operations and message-passing mechanisms). We have learned what we need to learn of operating system facilities from manuals and by trial-and-error. This has substantially slowed the development process. For example, it has taken us 14 months to finish the disk I/O subsystem (the development chronology has been given in Section 6.1). Before we have accomplished our task, we have wiped out the system disk on one of our PDP-11 systems three times! We have found most system manuals (such as the operating system manuals) are written for experienced people. Manuals with complete description of the facilities provided in a system and with various examples would be very helpful to systems programmers, especially the inexperienced programmers.

The choice of UNIX as a development environment has been a good one. Several software tools available on UNIX
have been helpful. We have used some of the tools extensively: YACC, LEX, LINT, MAKE and GREP. We feel that the use of these software tools has greatly reduced the implementation effort.

The run-time operating systems (VMS and RSX-11M) provide system features which are required for the implementation of MDBS. We have had to, however, use an overlay structure for some of the processes in the backends. The choice of RSX-11M-PLUS (instead of RSX-11M) would have alleviated the problem with overlaying a process, since this operating system provides a larger program-address space. We have not, however, originally known the difference between RSX-11M and RSX-11M-PLUS (another example of the problems caused by our lack of knowledge in operating systems).

The systems programming language, C, is simple enough for the implementors to learn and powerful enough to program the functionalities of MDBS. The tools provided under UNIX to C programmers have also been useful. Our primary problems in using C have been uninitialized variable references and out-of-bound array references. It would be helpful to have (for program development) a compiler and
run-time support system which detect these errors. Such features would greatly reduce program testing effort.

6.3.2. The Management Techniques: Effectiveness of Incremental Development and Formal Reviews, and the Need for Overall System Reviews

The techniques (in DBSIM) used for managing the software effort are: phased implementation, chief-programmer-team organization, and formal design and code reviews (see Chapter 3). The incremental development strategy has been effective. We believe that we could not have developed the complete system at once. Developing two versions of some MDBS functionalities, such as directory management, might have prolonged the development time, but we believe that it has been the right decision given the lack of experience of implementors.

The team organization has allowed different components of the system to be developed in parallel. This has made better use of the manpower and has reduced the development time. We have, however, overlooked the importance of team supervision. The management of a large-scale project, such as the MDBS implementation project, can affect the
development time. Such a supervision needs considerable time and attention. The MDBS group supervisor has had other duties beside managing the implementation effort. These duties include being chief programmer, backup programmer, and programmer in implementation teams. The supervisor would consider the management job as a secondary task and concentrate on the subprojects he is implementing. We now realize that the supervisor should have more time to manage the implementation effort and that he should consider the management job as his primary concern. It may even be cost and time-justified if the supervisor does not have any duty other than the team management. He should set milestones and supervise the team to meet the milestones.

The structured walkthroughs have been very effective and helpful. The reviews of the design and code by several people have resulted in a well-structured end product. Formal reviews have helped us to find almost all the design errors at the design stage of the development effort. Finding design errors at the design stage is, obviously, less costly than finding them at the coding or testing stages. Formal reviews have also helped us to find most of the coding errors at the coding stage of the development effort. The use of structured walkthroughs has also proved
to be a very useful way of sharing knowledge. The walkthroughs have, however, been useful only in finding errors in the individual components, such as a process, being presented. They have not helped with errors resulted from the impacts of different components on each other. This is because the individual components have been reviewed in the walkthroughs, and not the entire system. This is explained further in the following paragraph.

One procedure lacking in our management techniques is the concept of "periodic overall system review." We now realize that the entire system should be reviewed periodically in order to find inconsistencies and problems in the system at their earliest stages. The necessity for the directory-data concurrency control has been detected one and one half years after the implementation effort has started. This late detection has caused us to modify the code, which has already been completed, for the processes in the backends. This has resulted in loss of effort (we have had to modify the work that we have done in 6 months). Application of periodic overall system reviews might have detected our problem earlier and saved some time.
6.3.3. The Software Development Techniques

We have chosen techniques for the design, coding and testing phases of the software life cycle (see Chapter 4). (We have started the MDBS implementation using the requirements specification presented in [Hsia81a, Hsia81b], so we have not considered techniques for the requirements specification phase of the software life cycle.) In the following sections, we review the effectiveness of the development techniques used in DBSIM.

6.3.3.1. Design Techniques: Effectiveness of Use of Abstractions and a Systems Specification Language

The techniques (in DBSIM) used during the design stage of the software life cycle are: top-down design, use of abstractions, systems specification language and use of Jackson charts (see Chapter 4). The top-down design is effective in developing large systems. Furthermore, for our implementation effort, the top-down design has been more attractive since the high-level design for MDBS and possible decompositions are described in [Hsia81a, Hsia81b].

We have found use of abstractions very effective and useful. Use of abstractions has helped us to produce well-
structured programs. It has also made it easier to move from one MDBS version to another, and to change the implementation of different data structures in MDBS.

The use of a systems specification language has been helpful in communication among members. It has also reduced dependence on individuals since it provides complete and accurate documentation on the design. One problem in producing specifications for design is that of deciding how much detail should be included, i.e., what is "too much detail" and what is "not enough detail?" We have left this to the judgement of the reviewers in the walkthroughs. Mechanical details, such as checking that subscripts are within array bounds, are obviously too much details and should not be included in the specifications for the design. The programmers should, however, be cautioned to include these details when writing programs, and the reviewers should check this in code walkthroughs. Our original failure to check array bounds has resulted in spending considerable time to find some of the errors in programs during testing.

Jackson charts provide a two-dimensional representation of the design. They also contain fewer details than the
specifications since details such as variable declarations are not included in these charts. Jackson charts are more helpful than flowcharts since they reflect the structure of the design. These charts have made it easier for us to understand the design.

6.3.3.2. Coding Techniques: The Need for Coding Standards

In DBSIM the techniques used during the coding stage of the software life cycle are: principles of structured coding and program modification control (see Chapter 4). Structured coding is necessary to produce well-structured, easy-to-read and easy-to-understand programs. We adopted the principles of structured coding when we first started the project and since they have been effective, we continue to use them. We find our source code very readable. We doubt that adopting the principles of structured coding has resulted in inefficiencies in our programs. Our lack of systems programming experience might have, however, resulted in producing programs with some inefficiencies.

Large systems are usually programmed by a group of programmers, each programmer coding, say, a module. It is also the case that the original programmers usually leave the project and maintenance is done by other programmers.
We believe that standards for coding, such as indentation and variable/subroutine names, are essential for producing easy-to-read, easy-to-understand and easy-to-maintain programs. Our failure in setting standards for coding style has resulted in waste of manpower due to reformatting of source files. Lack of standards for choosing variable/subroutine names has created the problem of multiply-defined names when we have combined code written by different programmers. Several coding and naming standards would be acceptable. The issue is not deciding what standards to use. It is, rather, use of the same standards by all project members once the standards have been chosen.

Techniques for program modification control are needed when the source code is modified by more than one person. This is, of course, needed to avoid "update problem." The simple technique of having a list of file names currently being modified is easy and sufficient.

6.3.3.3. Testing Technique: Application of Unit, Integration and System Testing

In DBSIM the technique used during the testing stage of the software life cycle is: black-box testing (see Chapter 4). Black-box testing is appropriate when the function of
each component (a procedure, a process or the whole system) is well defined. Application of "unit testing", "integration testing" and "system testing" makes the testing phase more manageable.

We have done most of the testing with a view of the entire system as a "black box." We have used the four types of requests in MDBS (insert, retrieve, delete and update) as test data. We have applied unit testing and integration testing to some extent, but not completely. Fortunately, we have not run into major problems. This can be attributed to the use of walkthroughs for the design and code; most errors have been detected before testing has started. The testing strategy should have, however, been rigorously followed since it is a more practical approach to testing large systems such as MDBS.
7. CONCLUSION AND FUTURE WORK

In this report we describe an application of software engineering techniques to the development of large-scale database systems. A methodology, database systems implementation methodology (DBSIM), for such developments is proposed. This methodology and the results of using the methodology in the implementation of a multi-backend database system are presented. The methodology includes software engineering techniques for managing the database systems effort and for developing the database systems. The management techniques include a development strategy, a team organization and techniques for software quality assurance. The software development techniques include techniques for the design, coding and testing phases of the software life cycle. In this chapter we present the contributions of our work and directions for future work.

7.1. Contributions of the Present Work

We have experimented with the application of software engineering techniques to the development of large-scale
database systems. The methodology proposed and the results of an experiment are presented in this report. This is the first time to the knowledge of this author that a methodology for engineering database systems is proposed, and an experiment is carried out and reported.

The issues that must be addressed for a database system development effort are presented in Chapter 2. We categorize the issues as "management issues", "system hardware and software issues" and "database system implementation issues." These issues are studied in Chapters 3, 4 and 5. The aim of the study is to come up with the database systems implementation methodology (DBSIM).

The methodology, DBSIM, used in the MDBS implementation can be summarized as follows:
Management Techniques
 Incremental Development
 Chief-Programmer-Team Organization
 Formal Reviews for Design and Code
 Software Development Techniques
 Design Techniques
   Top-down Design and Use of Data and Service Abstractions
   Systems Specification Language
   Use of Jackson Chart
 Coding Techniques
   Principles of Structured Coding
   Program Modification Control
 Validation Techniques
   Black-box Testing Approach

This methodology is described in detail in Chapters 3 and 4. The results of our experiment are described in detail in Chapters 3, 4 and 5, and summarized in Chapter 6. Overall, the methodology has been effective and has worked well. Formal reviews for design and code, and use of abstractions have been most effective. The major points that we have overlooked are the need for overall system reviews and the need for coding standards.

Different versions of the multi-backend database system (MDBS) have been developed and tested [He83, Kerr83, Boyn83a, Boyn83b, Demu84]. This system is developed as a prototype for experimenting with the multi-backend software approach to database management. This approach configures the backends in a parallel way for performance improvement.
It also allows growth in the database and increase in the request rate without performance degradation and software complexity. The analytical studies (modeling techniques for performance evaluation) in [Hsia81a, Hsia81b] show that MDBS is promising for supporting large databases. By constructing working versions of MDBS, we are now able to carry out actual evaluation (measurement techniques for performance evaluation) of the system.

7.2. The Directions of Future Work

Our work is related to two areas: software engineering and database systems. So, we present directions for further research and experiment in each area.

7.2.1. Future Work in Software Engineering

Considerable work has been done in recent years on the subject of programming environments (software development systems). Further work is needed to provide a user-friendly programming environment. A user-friendly programming environment should help and direct the user in "thought" process and in all phases of software life cycle (requirements specification, design, coding and validation).
Tools and interfaces to aid the user in software development and document preparation need to be designed, implemented and integrated to provide a user-friendly programming environment.

The tools for the requirements specification phase of the software life cycle should guide and support the developer in the definition of software requirements. These tools should aid the developer in the simulation and analysis of the system to detect inconsistencies, contradictions, ambiguities and incompleteness in the specification. Since a requirements specification language should be more flexible than mathematical notations, a combination of a natural language and mathematical notations is needed in a requirements specification language. Thus, natural language processors would be helpful for this phase. The tools for requirements specification phase should also help in preparing test plans.

The tools for the design phase of the software life cycle should support different design methodologies. These tools should aid (and enforce) the development of a well-structured design. They should also help in finding inconsistencies. Tools to aid the developer in checking the
design against the requirements specification are also very useful. A design language and a graphical notation are needed for this phase. The language should allow the developer to specify the interface, effect and exceptional conditions of each module. The graphical notation should allow the developer to represent the program structures. Tools to analyze the design specifications are useful. Automated tools to generate the program structure charts from the design specifications are also useful.

Several tools are available for the coding phase of the software life cycle. These tools include language-oriented editors, static-code analyzers, debugging compilers and formatting aids. New tools for this phase are, however, required. Automated tools to generate the source code from the detailed design specification are useful. Tools for source code control are also useful. The use of such tools should result in a well-structured code. Such tools should also evaluate the complexity of a program (e.g., by assigning some measure to each language construct). This measurement can be used to keep the complexity of programs at a satisfactory (desirable) level.
The desirable tools for the validation phase of the software life cycle are correctness provers. Such tools verify that programs are correct by checking the validity of certain assertions about the programs. Since considerable research and work need to be done before such correctness provers can be developed for large-scale programs, tools for testing programs are more feasible for the near future. The tools for the testing phase should generate test data. These tools should also help the developer in finding and correcting errors.

More documents (such as the one presented in this report) on software development would be helpful to software engineers. The documents should include

- the methodology used in the development,
- effectiveness of each technique,
- experience (problems observed, recommendations and guidelines to software engineers, etc.), and
- facts and figures (design time, coding time, etc.).

These documents would help software engineers to better and more effective development of their system software.
7.2.2. Future Work in Database Systems

Details on the design, analysis and implementation of MDBS are reported in [Hsia81a, Hsia81b, He83, Kerr83, Boyn83a, Boyn83b, Demu84]. So, we have not repeated them in this report. We will, however, present direction for future work.

As previously mentioned, MDBS is promising for supporting large databases. Performance evaluation of the system needs to be carried out to verify this.

The data model used in MDBS, attribute-based data model [Hsia70], is a canonical model into which existing models (relational, hierarchical and network) can be transformed [Bane78a, Bane80, Bane78b]. Furthermore, the queries expressed in the data manipulation languages (DML) for these models can be translated to that used in MDBS. Detailed design and implementation of the two steps (database transformation and query translation) should be carried out.

A recovery mechanism is an integral part of a database system. Provision should be provided in MDBS for recovery from the following:
- Hardware failure. This includes a computer (the MDBS controller or one of the MDBS backends) failure, communication link failure, or a disk drive failure.

- Loss of a message in the system.

Records in MDBS database are grouped into clusters for efficient retrieval. For example, all the records for employees with salary greater-than-or-equal-to $10,000 and less-than-or-equal-to $20,000 may be grouped into one cluster. The clustering information (10000=<SALARY=<20000) is provided by the database creator. MDBS can be improved by designing mechanisms which collect information on the usage of the database. This information can be used at database reorganization time to better clustering of the records in the database.

7.3. Final Remarks on the Experiment

As described in Chapter 1, our group has two goals:

(1) Investigate the role of software engineering techniques in the implementation of large-scale database systems.

(2) Implement a prototype of the multi-backend database system for experimental use.

We have developed a database systems implementation
methodology (DBSIM) and have applied it to the development of MDBS. Our database systems engineering experience have been presented in this report. Thus, we have achieved our first goal. Since a prototype of MDBS is now available, we are able to begin the experimental studies on MDBS (our second goal). Our plan is to verify the simulation results on MDBS, and to investigate supporting existing data models (relational, hierarchical and network) on MDBS.


[DEC79a] "PCL11-B Parallel Communication Link Differential


