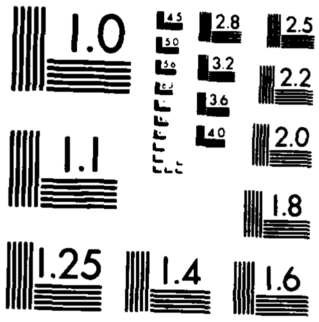
AD-A141 786 PERFORMANCE EVALUATION TOOLS FOR A MULTI-BACKEND DATABASE SYSTEM(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA J G KOVALCHIK DEC 83 1/1

UNCLASSIFIED F/G 9/2 NL

END
DATE
FILMED
7-84
DTIC



2

AD-A141 786

NAVAL POSTGRADUATE SCHOOL
Monterey, California

DTIC
ELECTE
JUN 4 1984
S D
B

THESIS

PERFORMANCE EVALUATION TOOLS FOR A MULTI-BACKEND DATABASE SYSTEM

by

Joseph G. Kovalchik

December 1983

Thesis Advisor: David K. Hsiao

Approved for public release; distribution unlimited

84 06 04 028

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A141786	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Performance Evaluation Tools for a Multi-backend Database System		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis December, 1983
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Joseph G. Kovalchik		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		12. REPORT DATE December, 1983
		13. NUMBER OF PAGES 68
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

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

18. SUPPLEMENTARY NOTES

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

Performance Evaluation, Backend Database Machine, Database, MDBS.

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

In this thesis, we discuss the development of the necessary tools for the performance evaluation of a multi-backend database system known as MDBS. The basic motivation of the multi-backend database system (MDBS) is to develop an architecture which spreads the work of the database system among multiple backends. It is a major aim of this system to allow capacity growth by the use of additional disk drives and performance improvement by the use of additional backends. However, to verify the (Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT (Continued)

design and implementation, it is necessary to test the capability of MDBS in capacity growth and performance gain.

Three tools for the performance and capacity tests are investigated. The first tool is the file generation package which creates test files for any artificial database. The second tool is the database load subsystem which loads the artificial database into MDBS. The third tool is the request generation package. This package creates test requests to query MDBS.

The following methodology is used to create an effective tool. First, the properties of an ideal tool are described. Then available existing programs are reviewed and evaluated to determine which program best meets the desired features. Lastly, the programs are upgraded to ensure that they are compatible with the current implementation, and meet the desired features.

The main goal is to develop the necessary tools to generate tests in measuring the extensibility of MDBS, i.e., how does MDBS perform as more backends are added? Performance is expected to improve (maintain) as the number (size) of the backends (database) is increased.

Accession For	
NTIS GRA&I	✓
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

Performance Evaluation Tools for a Multi-backend Database System

by

Joseph G. Kovalchik
Lieutenant, United States Navy
B.S. , United States Naval Academy, 1977

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1983

Author: Joseph G. Kovalchik

Approved by: David K. Hsiao
Thesis Advisor

Douglas S. Kerr
Second Reader

David K. Hsiao
Chairman, Department of Computer Science

K. T. Marshall
Dean of Information and Policy Sciences

ABSTRACT

In this thesis, we discuss the development of the necessary tools for the performance evaluation of a multi-backend database system, known as MDBS. The basic motivation of the mutlti-backend database system (MDBS) is to develop an architecture which spreads the work of the database system among multiple backends. It is a major aim of this system to allow capacity growth by the use of additional disk drives and performance improvement by the use of additional backends. However, to verify the design and implementation, it is necessary to test the capability of MDBS in capacity growth and performance gain.

Three tools for the performance and capacity tests are investigated. The first tool is the file generation package which creates test files for an artificial database. The second tool is the database load subsystem which loads the artificial database into MDBS. The third tool is the request generation package. This package creates test requests to query MDBS.

The following methodology is used to create an effective tool. First, the properties of an ideal tool are described. Then available existing programs are reviewed and evaluated to determine which program best meets the desired features. Lastly, the programs are upgraded to ensure that they are compatible with the current implementation, and meet the desired features.

The main goal is to develop the necessary tools to generate tests in measuring the extensibility of MDBS, i.e., how does MDBS perform as more backends are added? Performance is expected to improve (maintain) as the number (size) of the backends (database) is increased.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENT

This work is supported by Contract N00014-75-C-0573 from the Office of Naval Research to Dr. David K. Hsiao and conducted in the Laboratory for Database Systems Research, located at the Naval Postgraduate School. Dr. Douglas S. Kerr, Adjunct Professor of Computer Science, is the present Director of the Laboratory. The laboratory equipment is supported by DEC, ONR and NPS.

I would like to thank all those who have supported the MDBS project and who have contributed to the development of this document. In particular, the design and analysis were developed by Jai Menon. Drs. Hsiao and Kerr have provided much of the guidance needed in developing the project. Their time and patience proved invaluable in the development of the implementation of the performance tools. Earlier work accomplished in the field of performance evaluation of database machines infuenced the development of the tools presented here. This work was accomplished by Paula Strawser, Michael Crocker, Curtis Ryder, Robert Bogdanowicz, and Vincent Stone. Special thanks are also extended to Al Wong, who as a member of the Computer Science professional staff provided many services to the laboratory. Lastly, I would like to thank Ruth Olson, who had provided many editorial comments concerning this document.

I. AN INTRODUCTION

This chapter presents a brief review of the multi-backend database system (MDBS). First, the physical arrangement of MDBS is presented. This is followed by a presentation of the process structure of MDBS. Lastly, the actions taken in servicing requests, both insert and non-insert requests, are reviewed. References are cited for the interested reader in order to gain a more detailed understanding of MDBS.

A. THE MULTI-BACKEND DATABASE SYSTEM

The multi-backend database system (MDBS) uses one mini-computer as the master or controller, and a varying number of minicomputers and their disks as slaves or backends. MDBS is designed to provide database growth and performance enhancement by the addition of identical backends. No special hardware is required. The backends are configured in a parallel fashion. A new backend may be added by simply replicating the existing software on the new backend, thus avoiding reprogramming efforts. A prototype MDBS has been completed in order to carry out the design verification and performance evaluation developed in [Ref. 1] and [Ref. 2]. The implementation efforts are described in [Ref. 3] through [Ref. 5].

The equipment configuration of the system is shown in Figure 1.1. The host computer is connected to MDBS through the controller. The backends are connected to the controller through a broadcast bus. When the controller receives a request from the host, it delivers the request to all backends simultaneously over the broadcast bus.

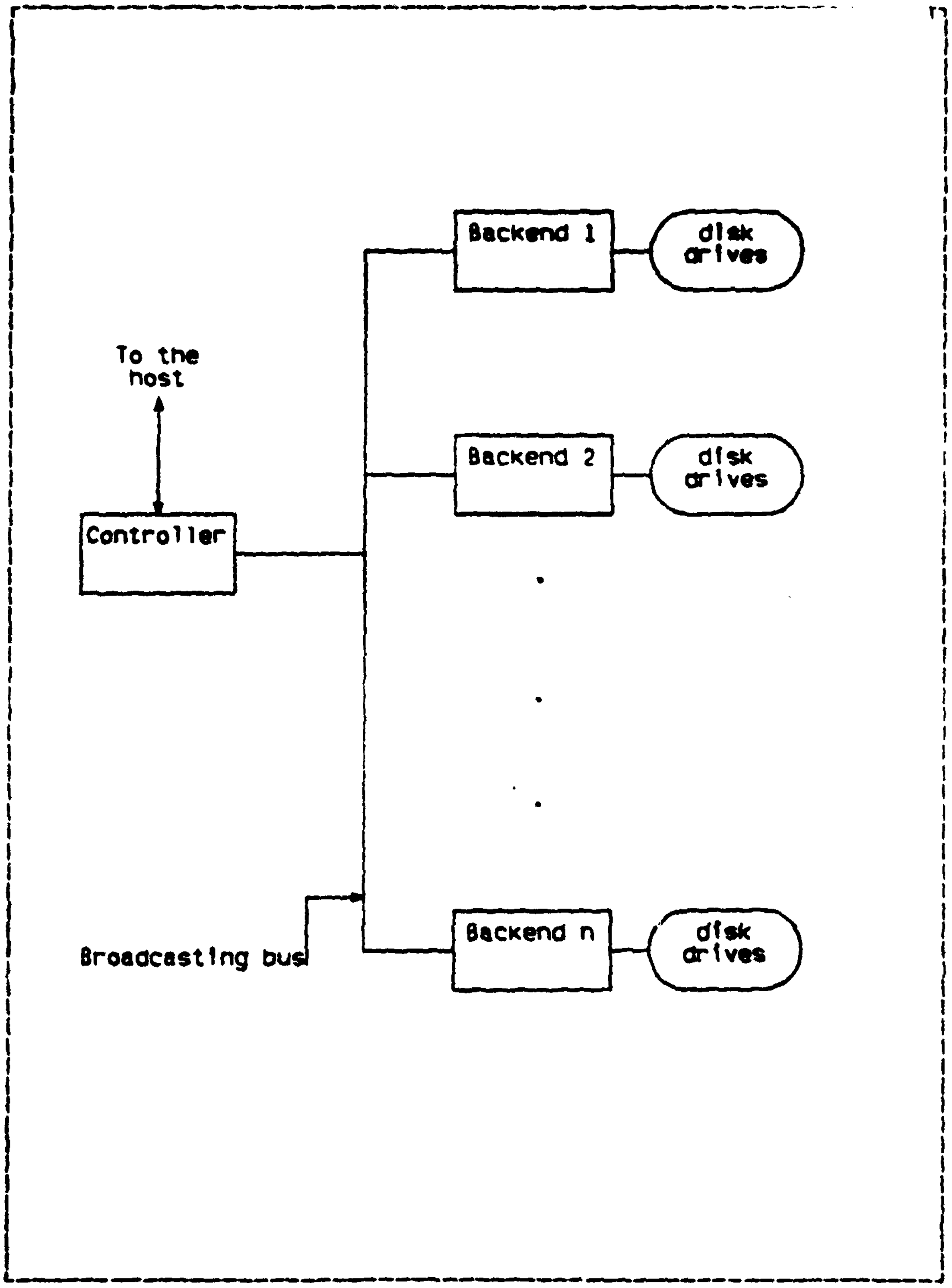


Figure 1.1 HARDWARE CONFIGURATION OF MDBS.

Since the data is distributed across all backends, all backends can execute a request in parallel.

The division of labor between the controller and the backends is illustrated through the process structure of Figure 1.2. The MDBS controller handles three functions. The <u>request preparation function</u> prepares a request for transmission to the backends. The <u>insert information generation function</u> processes the insert requests which require additional information used by the backends. The <u>post processing function</u> handles the work necessary when the replies are returned to the controller from the backends but before reaching the host.

The backends in MDBS carry out three different functions. The <u>directory management function</u> performs descriptor search, cluster search, address generation, and directory table maintenance. The <u>record processing function</u> performs record storage, record retrieval, record selection, and attribute-value extraction of the retrieved records. The <u>concurrency control function</u> performs operations to ensure that the concurrent and interleaved execution of the user requests will keep the database consistent.

Before proceeding to describe the sequence of actions required during a request servicing, some terminology is presented as a review. The smallest unit of data is a keyword, which is an attribute-value pair. Information is stored in terms of records, which are made up of keywords and a record body. A predicate is of the form (attribute, relational operator, value). A query is any Boolean expression of predicates. Records are logically grouped into clusters based on the attribute values and the attribute-value ranges in the records. Internally, the values and value ranges are called descriptors. For the user, these attribute values are termed keywords. Each descriptor is identified by a descriptor id to save computing time and

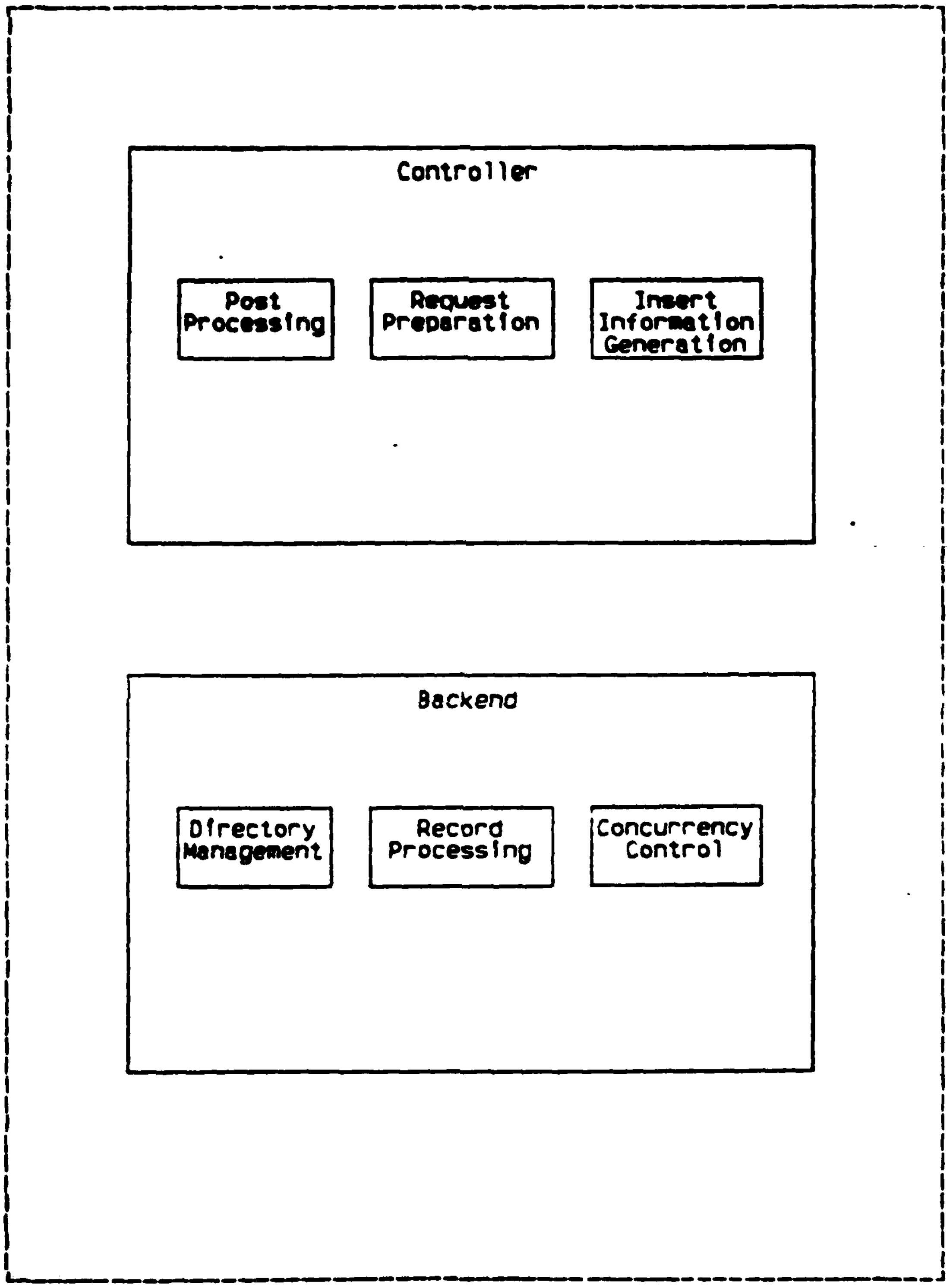


Figure 1.2 PROCESS STRUCTURE OF MDBS.

memory space. A prespecified set of requests is referred to as a transaction.

B. REQUEST EXECUTION

This section describes the sequence of actions taken by MDBS in carrying out a request. First, the insert request will be discussed. Then the non-insert requests will be described. Non-insert requests are requests for deletion, retrieval, or update.

1. Actions for Insert Requests

The sequence of actions for an insert request is shown in Figure 1.3. A request from the host machine enters the Request Preparation process. Request Preparation broadcasts the number of requests in the transaction to Post processing in order to determine when a transaction is completed. Request Preparation may send an error to Post Processing if there is a syntax error in the request. When a transaction is completed Post Processing sends the results to the host machine. Request Preparation then broadcasts the request to Directory Management. Each backend finds the descriptor ids associated with the request. The backends then exchange descriptor id information.

After receiving the descriptor ids from the other backends, Directory Management sends the cluster id to Insert Information Generation. Insert Information Generation then determines which backend is to store the record. The selected backend determines the address of the new record and stores it. The other backends discard the record. Finally, Record Processing sends an action-completed message to Post Processing, which in turn informs the host.

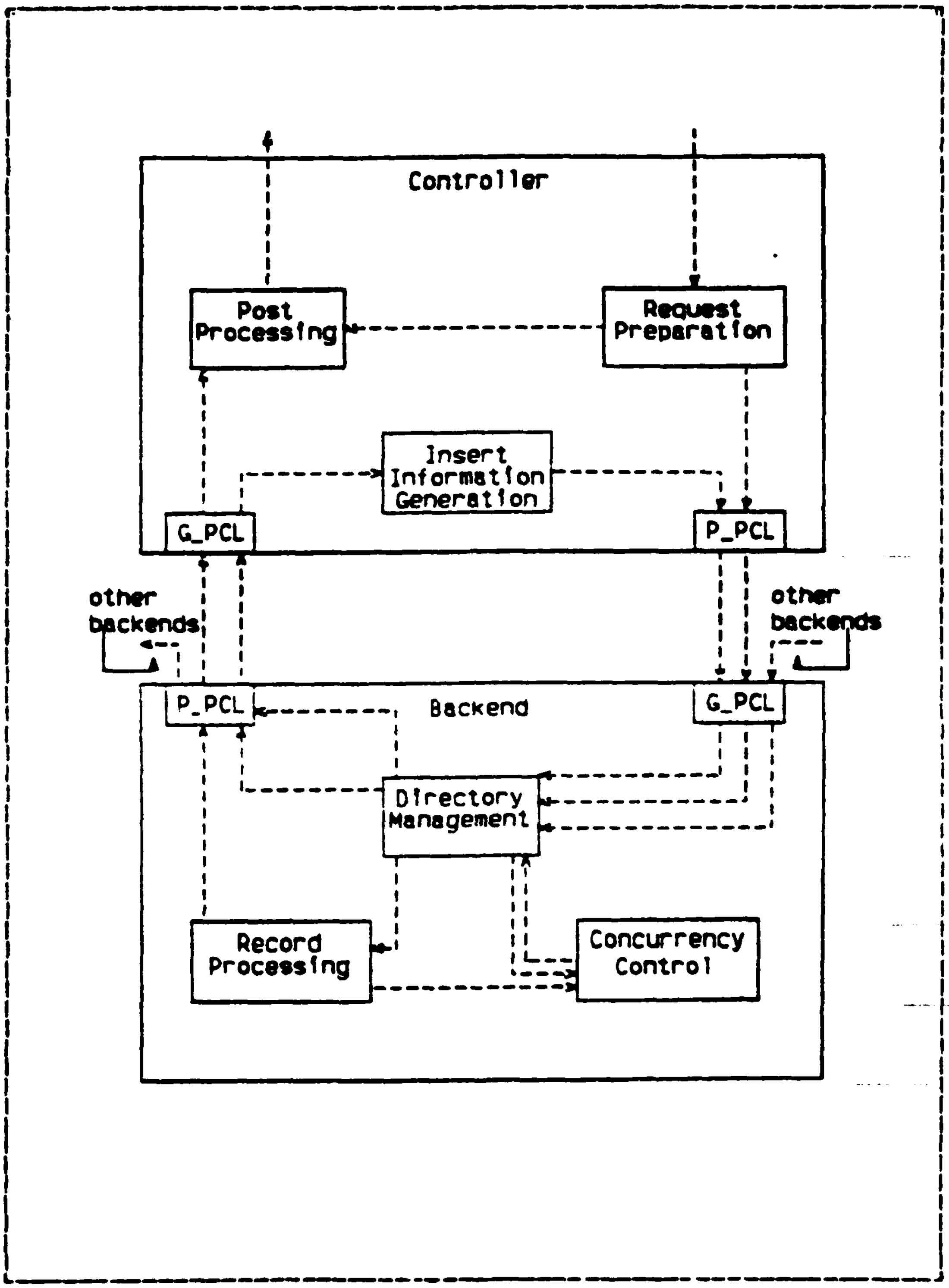


Figure 1.3 SEQUENCE OF ACTIONS FOR AN INSERT REQUEST.

2. Actions for Non-insert Requests

The sequence of actions for a non-insert request is shown in Figure 1.4. The actions for a retrieve will be discussed only, since the other types of requests are quite similar. A request from the host machine enters the Request Preparation process. Request Preparation sends the number of requests in the transaction to Post Processing in order to determine when a transaction is completed. Request Preparation may send an error to Post Processing if there is a syntax error in the request. When a transaction is completed, Post Processing sends the results to the host machine. Request Preparation then broadcasts the request to Directory management. Each backend finds the descriptor ids associated with the request. The backends then exchange descriptor id information.

After receiving the descriptor ids from the other backends, Directory Management determines the cluster ids. Lastly, Directory Management determines the addresses of the records of the identified clusters. Record Processing gets the records from secondary storage and extracts the necessary information. If aggregate operators, for example, the average, are specified in the retrieve request, they are applied at this time. The partially aggregated values are sent to Post Processing. Post Processing sends the results to the host following any further aggregate operations.

This concludes the review of MDBS. Attention is now turned toward performance issues of this system in the following chapter.

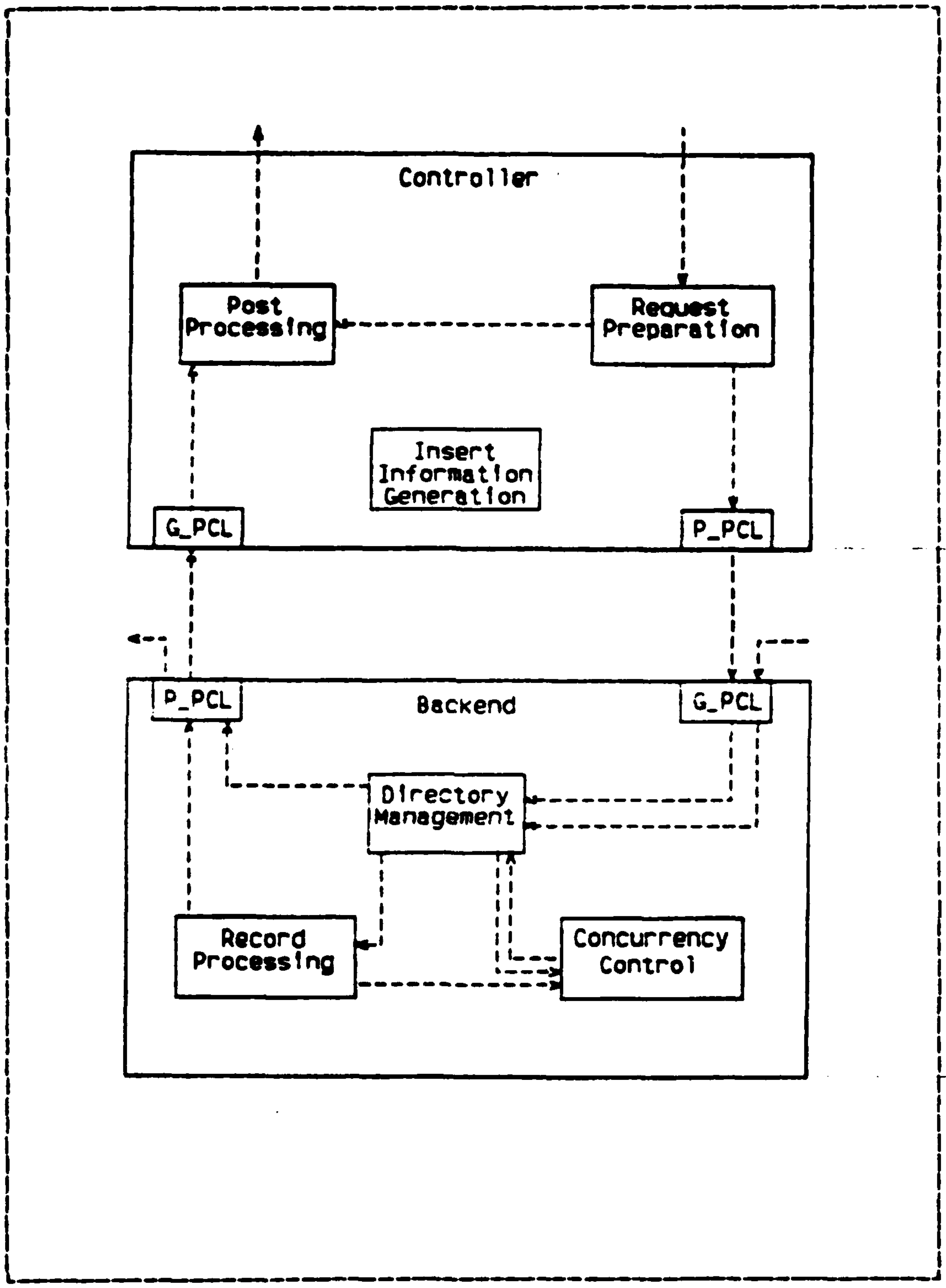


Figure 1.4 SEQUENCE OF ACTIONS FOR A NON-INSERT REQUEST.

II. PERFORMANCE EVALUATION

A. TWO VIEWS OF PERFORMANCE MEASUREMENT

Now that the MDBS has been described, it is reasonable to ask "how does one determine the performance of such a system?" There are two viewpoints of performance evaluation. The first is the macroscopic viewpoint in which the key performance measurement is the relative response time. The second viewpoint is the microscopic viewpoint. This viewpoint is concerned with measuring the times needed to perform various subtasks which are carried out in servicing a request. In [Ref. 6], the motivation for the macroscopic measurement is provided. This chapter is concerned with describing the performance issues which arise when using the macroscopic viewpoint. Thus in testing the MDBS, the macroscopic viewpoint will be used before proceeding to the microscopic viewpoint.

B. CRITERIA FOR PERFORMANCE EVALUATION AND TOOL SELECTION

1. Macroscopic Viewpoint

As stated above, with the macroscopic viewpoint the key performance measurement is the relative response time. That is, the concern lies mainly with the affect of various changes to the system on the response time. These changes and therefore their relative response times are prompted by the variables described in the following section.

2. Performance Issues

The macroscopic viewpoint is concerned with changing four categories of variables and observing their affect on the relative response time. These variables include system configuration variables, cluster formation variables, request construction variables, and storage variables.

The system configuration variables deal with the following questions on how MDBS performs when: the number of backends remains constant but the database increases, the database remains constant and the number of backends increases, the number of concurrent users increases, the number of requests per transaction increases, and the presence of concurrency control is measured against the absence of concurrency control.

The cluster formation variables deal with the following questions on how MDBS performs when: the number of descriptors on any attribute increases, the average size of clusters in the database ranges over small, medium, and large size, and the number of attributes and thus the size of the attribute table increases.

The request construction variables deal with the following questions on how MDBS performs when: the request makeup is retrieve-intensive vs. update-intensive, the complexity of the query increases, the relative mix of query types is varied, the retrieved information is either a projection of the record or the whole record, the query predicates are permuted, and the request uses either non-directory keywords or directory keywords.

Lastly, the storage variables deal with the following questions on how MDBS performs when: the data placement strategy of the database changes, the tuple width increases, and the size of the retrieved information exceeds that available in the main memory.

Thus it can be seen that several variables influence the performance of MDBS. This is not an all-inclusive list. However, the list will serve as a basis for developing the desired properties of each performance tool. Each tool will be discussed along with its desired properties in the following sections.

C. DESIRABLE PROPERTIES OF THE TEST FILE GENERATION PACKAGE

The purpose of the file generation package is to create an artificial database which will eventually be loaded into MDBS. This is the first tool to be used for the evaluation. Several parameters are likely to be varied in the light of the performance issues. Their desired properties are as follows. The input parameters to such a package may include: file size in number of records per file, attribute-value size in bytes of storage, record size in number of attributes values, data types of attribute values, and database size in number of files per database. In addition, parameters must indicate whether values of attributes are taken from random functions, or from predetermined sets, and whether uniqueness of values is desired.

D. DESIRABLE PROPERTIES OF THE DATABASE LOAD SUBSYSTEM

The database load subsystem is responsible for taking the files created by the file generation package and for properly loading the files into MDBS. In the process of loading the database, the database load subsystem must also create the necessary tables used in directory management.

The database load subsystem must be designed so that the performance evaluation may utilize various cluster formation variables and storage variables with minimum effort. The cluster formation variables and storage variables with which the performance may be concerned include the following. The performance may be expected to depend upon whether the number of descriptors (attributes) is large or small. Certainly, when entering a large number of descriptors (attributes), the chance for error in this menial task is great. Therefore, the ease of specifying the descriptors (attributes) must be guaranteed. The variation of cluster size may affect performance. The cluster size is a function of the number of descriptors, the size of the input files, and the values used in the attribute fields. Therefore, these three parameters should be entered independently. The data placement strategy, i.e., how records are distributed across the backends, also affects performance. While simulation studies described in [Ref. 1] and [Ref. 2] show that the track-splitting-with-random-placement strategy is the most desirable, the ability to change the placement strategy will provide a means of confirming these studies.

E. DESIRABLE PROPERTIES OF THE REQUEST GENERATION PACKAGE

The request generation package is concerned with creating and executing test requests. The request formation variables will be altered by the performance evaluation team in this performance evaluation tool. The request formation variables will be changed in order to vary the following: the percentage of the types of requests (retrieve, update, insert, or delete), the percentage of aggregate operators (ave, max, min, sum, and count) in retrieve requests, the complexity of the request query (A simple query will consist of one to two predicates, and a complex query will consist of ten to fifteen predicates), the order of the predicates appearing in the request, and the number of attributes to be projected in the retrieve request.

The request generation package must also possess the ability to allow the following: vary the length of the transaction to determine its effect on system performance, tag requests with user identification in order to test concurrency control, retrieval of a record defined over the null descriptor, execute a retrieve request where the entire cluster is stored at one backend, and compare the above performance with a retrieve request where the cluster is distributed across all backends.

It is now appropriate to proceed to the details of each of the above three tools. In the following chapter the test file generation package is discussed. Chapter IV deals with the details of the database load subsystem, and Chapter V develops the test request generation package.

III. THE TEST FILE GENERATION PACKAGE

In this chapter, we discuss the test file generation package development. In the first two sections, we review the purpose and desired properties of the package. In the next two sections, we discuss how the basic program was selected from existing file generation tools. Finally, in the last two sections we discuss the upgrading of the selected program and future enhancements which will further aid the performance evaluation team.

A. THE PURPOSE

The first set of performance evaluation experiments will use test data which is generated by a program in the form as specified by the experimenter. This process may be viewed in three steps. The first step consists of defining the structure of the files to be generated. The second step determines where the values for the specified attributes will be generated. Lastly, the files are generated and stored for future use.

B. DESIRED PROPERTIES

The input parameters to such a package may include: file size in number of records per file, attribute size in bytes of storage, record size in number of attribute values, data types of attributes, database size in number of files per database, whether values of attributes are taken from random functions or are selected from predetermined sets, and whether uniqueness of values is desired.

C. EXISTING PROGRAMS

Two programs were reviewed in order to determine which possesses the largest number of desired properties and still would require the least effort to ensure system compatibility with the current version of MDBS. The first of the twc programs was originally designed in [Ref. 3]. The second was a latter attempt to simplify the test file generation package.

1. The Original Test File Generation Package

In this program the test data is generated and stored in files. Several characteristcs of the file are specified by the experimenter. Each file is given a name. The data in the records is specified in a fixed number of attribute-value pairs. The type of data in the attributes is integer, string, and floating-point numbers. These values are generated in either predetermined files, called sets, created by the experimenter, or are randomly generated by separate functions. Only a uniform distribution of the various data types is available. This program contains all of the desired properties stated above, except the ability to guarantee uniqueness of the records created.

2. The Shortened Test File Generation Package

This program was written in order to reduce the complexity of the original test file generation package. Many of the features of the original program remain intact. Two important differences exist. The shortened version only allows the use of predetermined sets of values to be used, therefore not allowing randomly generated values. The second difference is the fact that the files generated must be of length of less than or equal to 10,000 records. An advantage of the shortened version is that it is combined with the shortened database load program, which is discussed in the following section.

D. SELECTION OF THE TEST FILE GENERATION PACKAGE

The shortened version of the test file generation package was selected initially as the file generation tool. MDBS is currently undergoing a change in the version of the compiler used. In an attempt to keep the conversion of MDBS simple, the shortened version was chosen. This version allowed a rapid conversion. However, only user defined sets of values are selected for the attribute values. This is considered a disadvantage. Perhaps the overriding consideration in the selection of the shortened version was the fact that its associated database load subsystem was much simplier. The discussion of this subsystem is provided in detail in the following section.

E. THE UPGRADING PROCESS

The upgrading process for the shortened version of the test file generation package was relatively simple. The C compiler originally used in the implementation was an older version. The new version is being used by MDBS. Several minor compiler differences with respect to acceptable syntax were rapidly fixed.

F. FUTURE IMPROVEMENTS

Because the shortened version possesses all but one of the desired properies discussed in chapter II, only one future change is anticipated.

Two approaches which provide the shortened version with the capability of randomly generating values exist. The first of these alternatives includes adding the functions to the program with the additional user interface to select these as options. The second alternative is to adapt the original test file generation package to be compatible with the shortened database load. The task would be simplified by choosing the first alternative.

This concludes the discussion of the test file generation tool. In the following chapter, we discuss the properties of the selected database load subsystem.

IV. THE DATABASE LOAD SUBSYSTEM

In this chapter, we discuss the database load subsystem development. In the first two sections, we review the purpose and desired properties of the subsystem. In the next two sections, we discuss how the basic program was selected from existing database load tools. Finally, in the last two sections, we discuss the upgrading of the selected program and future enhancements which will further aid the performance evaluation team.

A. THE PURPOSE

The database load subsystem is a software tool used to designate an input source file and to create a database from that source file. It also allows several related files to be consolidated into one database if desired. The first phase in the database load subsystem is to define the input files and the database. The second phase consists of constructing various directory management tables. Lastly, the records are distributed across the backends.

B. DESIRED PROPERTIES

The database load subsystem must be designed so that the performance evaluation may utilize various cluster formation variables and storage variables with minimum effort. The performance may be expected to depend upon whether the number of descriptors(attributes) is large or small. The ease of specifying the descriptors (attributes) must be guaranteed. The variation of cluster size may affect performance. The cluster size is a function of the number of descriptors, the size of the input files, and the values used in the attribute fields. These three parameters should be entered independently. The data placement strategy, i.e., how records are distributed across the backends, also affects performance. The ability to change the placement strategy will provide a means of confirming simulation studies.

C. EXISTING PROGRAMS

Two database load subsystems were reviewed. In this section the merits of both of the existing programs are discussed. The original database load subsystem is covered first, then a shortened version of the database load subsystem is evaluated.

1. The Original Database Load Subsystem

The original database load subsystem was first designed at the beginning of the implementation stage of MDBS. The process is viewed as four logical phases. The first phase is the database definition phase, in which the user specifies various characteristics of existing source files and the characteristics of the database to be created. The second phase is the record preparation phase, in which the data is read from the input files and prepared for loading. The third phase is the record clustering phase, in which the prepared records are sorted into clusters. The last phase is the record and table distribution phase. This phase distributes the records and the directory management tables to the backends.

2. The Shortened Database Load Subsystem

As stated in Chapter II, the shortened database load subsystem is much simpler than the original database load subsystem. This implementation can be viewed as two phases.

The first phase is the directory table construction phase, in which specified database parameters are read from existing files and the directory tables are constructed. The second phase is the record distribution phase. In this phase the records are distributed to the backends by using insert requests. Thus this subsystem uses currently existing directory management functions to load the database records.

D. THE SELECTION OF THE DATABASE LOAD SUBSYSTEM

Several disadvantages to the original database load program exist. Since it was created at the inception of MDBS design, it possessed many system incompatibilities with the current version of MDBS. Once again the large size of the program posed a significant maintenance problem with respect to the conversion of the system to the new compiler. For these reasons this program was not selected.

The shortened version of the database load subsystem was chosen as the basis for the database load tool. This was due to the fact that it used existing directory management code and that it was much simpler to understand and thus maintain.

E. THE UPGRADING PROCESS

In this section, we now discuss the upgrading of the shortened version of the database load subsystem. A discussion of the communication among processes is presented. Then the changes to the database load subsystem are discussed.

1. Message Passing

In order to load the current version of MDBS, it is necessary to change the database load subsystem so that it could communicate with the backend process of directory management. The database load subsystem is implemented as a separate process in the controller. A brief discussion of message passing in MDBS is presented below.

a. Message Passing Within a Backend

The backends are supported by PDP-11/44s running under RSX-11M operating system. The inter-process-communication facility is the shared access to physical memory. Suppose process X wants to send a message to process Y. X will copy the message into the shared area. Then X tells the operating system to send the address of the message to process Y. When Y is ready to receive a message, it gets the address of the message from the operating system's queue of such addresses. Process Y then copies the message into its own memory space.

b. Message Passing Within the Controller

The MDBS controller is a VAX-11/780 using the VMS operating system. The inter-process communication facility is the mailbox. The mailbox is a software input/output device. If process X wishes to send process Y a message, process X first issues a send command to process Y's mailbox. When process Y issues the read command on its mailbox it will be given the message sent by process X. The mailbox can queue several messages.

c. Message Passing Between Computers

Communication between computers in MDBS is achieved by using a time-division-multiplexed bus called the parallel communication link (PCL). Two interface processes to the PCL are used in each computer. The first process, called put_PCL, puts the message to be sent to the other computers on the PCL. The second process, called get_PCL, receives the message from the bus and then passes the message to the appropriate process. PCLs are presently used to simulate the broadcast bus and will be replaced physically by a broadcasting bus later.

2. Directory Tables

Several directory tables exist in order to process requests. In this section the logical descriptions of such tables are discussed. This will allow some insight into what kind of messages must be sent during the loading of the database.

The Attribute Table (AT) contains a list of the directory attributes and a pointer to the descriptors defined on these attributes. The AT is located at each backend. The Descriptor-to-Descriptor-Id (DDIT) Table contains the descriptors and their corresponding descriptor ids. Each section of the DDIT is associated with a directory attribute and contains the descriptors defined on that attribute. The DDIT is located at each backend. Since type-C sub-descriptors are created dynamically as new records are inserted, the type-C attributes must be recorded in a table called the Type-C-Descriptor-Table (TCDT). The TCDT is located in the controller. When an insert request contains a record with a type-C attribute and the value of the attribute does not appear in a type-C descriptor, a new type-C descriptor will be created by the Insert Information

Generation process. This process will then record the descriptor in the TCDT. Thus all directory attributes and their corresponding descriptors are sent to the backend's Directory Management processes. All type-C attributes are also sent to the Insert Information Generation process in the controller.

3. Specific Upgrades

The database load subsystem program was changed by allowing it to communicate with the backends in order to load the database to the backends. In order to distribute the directory management tables to all backends, the database load subsystem must be given its own mailbox and access to the directory management physical areas located in the backends. All of the functions which create the directory management tables were moved to the backends and appropriately placed in the directory management processes. Data necessary to construct these tables was passed to the backends by using messages containing codes which indicate the type of action to be taken. Because the backends can construct the tables in parallel, this did not significantly burden the database load process. In order to support the message passing ability, send and receive routines specific to the database load process were written. Figure 4.1 illustrates the inter-process communication involved with the directory table construction phase.

In order to load the records into the database, communication between the request preparation process (located in the controller) and the database load subsystem was established. This allowed the database load subsystem to send the insert requests directly to request preparation. Thus the database load subsystem was given access to the request preparation mailbox. It was also necessary to send the Insert Information Generation process all of the type-C

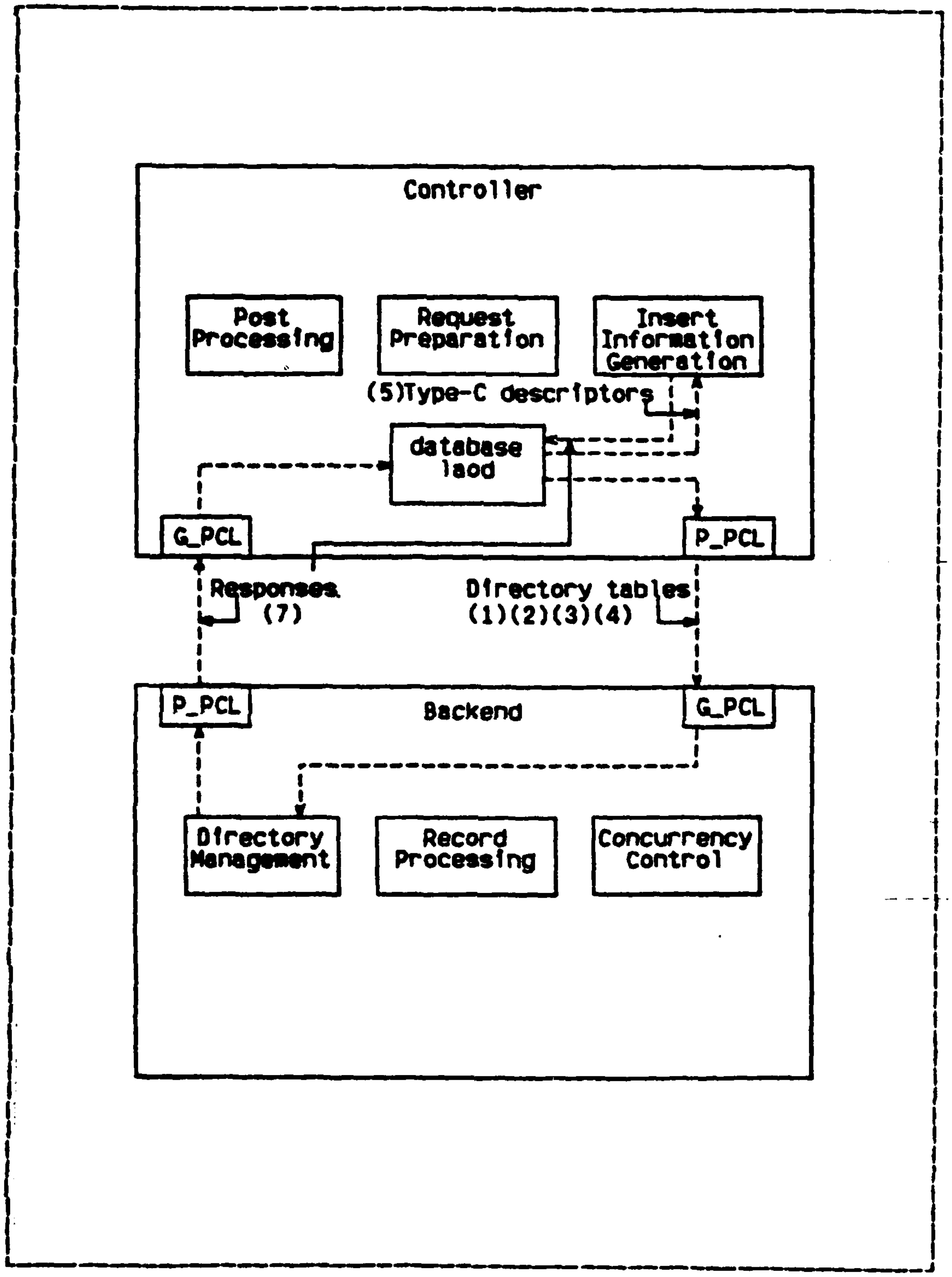


Figure 4.1 COMMUNICATIONS: DIRECTORY TABLE CONSTRUCTION.

attributes for insertion into the TCDT. Figure 4.2 shows the inter-process communication of the record distribution phase.

The following is a summary of the types of messages which were added to the database load subsystem:

Message type:	(1) Create AT
Source:	Database Load (DBL)
Destination:	Directory Management
Explanation:	This message creates an AT for the given database name.
Message type:	(2) Add Attribute to AT
Source:	Database Load (DBL)
Destination:	Directory Management
Explanation:	This message adds an attribute to the AT for the given database.
Message type:	(3) Add Descriptor to DDIT
Source:	Database Load (DBL)
Destination:	Directory Management
Explanation:	This message adds a descriptor to the DDIT for the given database.
Message type:	(4) Add the end of descriptor flag
Source:	Database Load (DBL)
Destination:	Directory Management
Explanation:	This message adds the flag to signal the end of the descriptor list.
Message type:	(5) Load type-C
Source:	Database Load (DBL)
Destination:	Insert Information Generation
Explanation:	This message passes the type-C attribute to IIG for entry into the TCDT.

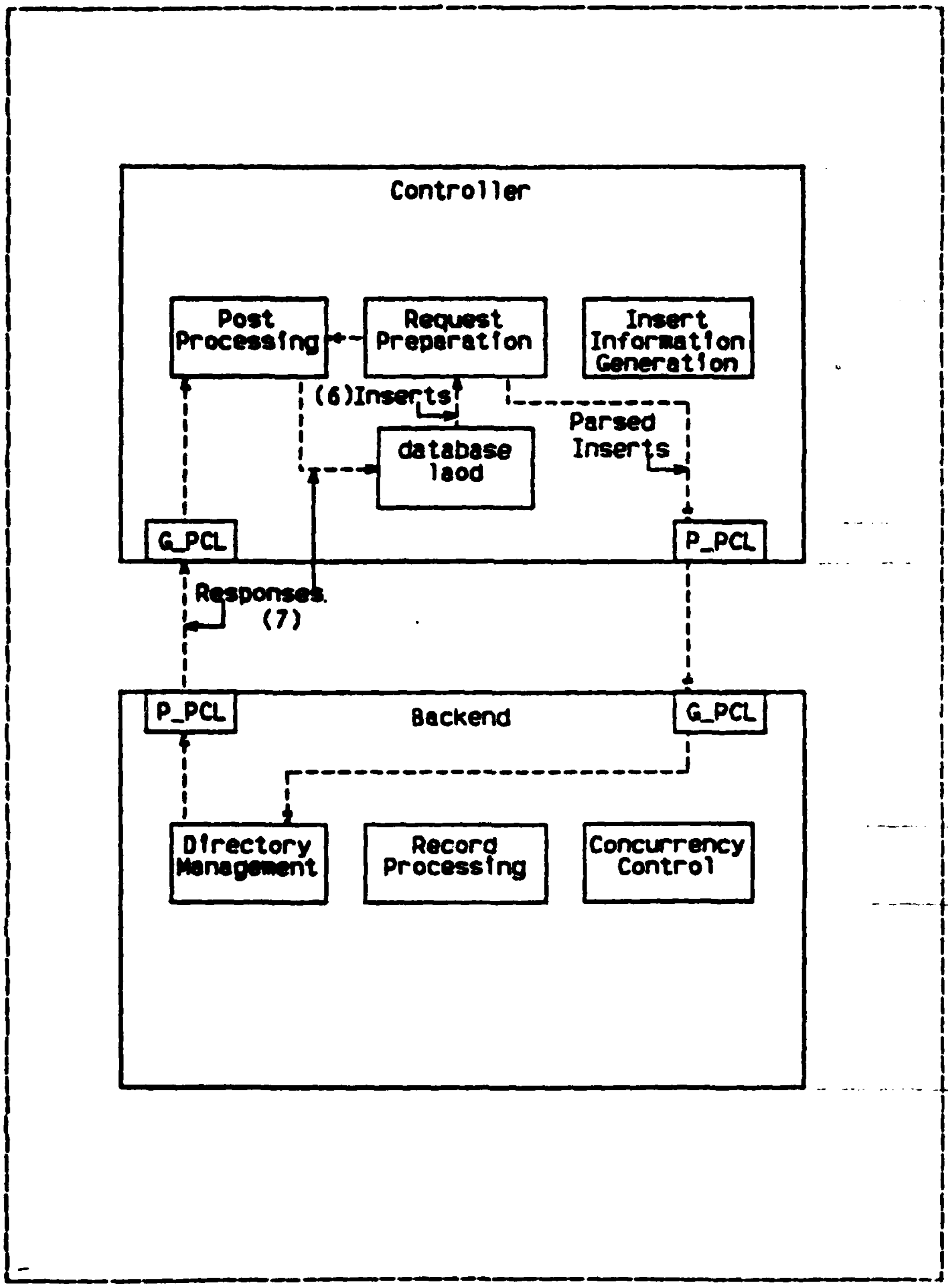


Figure 4.2 COMMUNICATIONS: RECORD LOADING.

Message type:	(6) Insert record
Source:	Database Load (DBL)
Destination:	Request Preparation
Explanation:	This message sends the record to be loaded to RP.

Message type:	(7) Responses
Source:	Directory Management and Insert Information Generation
Destination:	Database Load
Explanation:	This group of messages informs DBL of action that is actually carried out as requested by the above messages from DBL. They also include error messages.

Thus for each of the messages (1) through (6), a type (7) message is sent to the Database load subsystem. This concludes the upgrading of the database load subsystem.

F. FUTURE IMPROVEMENTS

The database load subsystem contains all of the desired properties discussed above with the exception of the ability to change the data placement strategy. Due to the manner in which the database is loaded, this would require a change in the directory management process. Further research is required to investigate the ramifications of changing the directory management process. This feature should be delayed until the system conversion to the new compiler is completed.

V. THE TEST REQUEST GENERATION PACKAGE

In this chapter, we discuss the test request generation package development. In the first two sections, we review the purpose and desired properties of the package. In the next two sections, we discuss how the basic program was selected from existing request generation tools. Finally, in the last two sections, we discuss the upgrading of the selected program and future enhancements which will further aid the performance evaluation team.

A. THE PURPOSE

The purpose of the test request generation package is to provide an easy means of creating a list of test requests which will be executed in order to test MDBS. The package also aids the evaluation team in executing the list of requests. The list of requests are saved in a file for future use, in order to avoid regenerating the list of requests.

B. DESIRED PROPERTIES

Recall that the test request generation package permits the request formation variables to be altered by the evaluation team. This allows the following to be varied: the percentage of the types of requests (retrieve, update, insert, or delete), the percentage of aggregate operators (ave, max, min, sum, and count) in retrieve requests, the complexity of the request query, the order of the predicates appearing in the request, and the number of attributes to be projected in the retrieve request.

The request generation package must also possess the ability to allow the following modifications: vary the length of the transaction, tag requests with user identification, retrieve a record defined over the null descriptor, and execute a retrieve request in which the entire cluster is stored at one backend and compare the performance with a request which retrieves records from a cluster which is stored across all backends.

C. EXISTING PROGRAMS

Two existing programs were reviewed in order to select the one which best fits the desired properties and is compatibile with the current version of MDBS. Both programs implement the test request generation package in the controller. The next section discusses version A of the test request generation package. Version A was originally designed at the commencement of the implementation of MDBS. Version B was a later version.

1. Version A

Version A may be described as a package which aids the user in developing a list of requests. The user is guided through the construction of one request at a time. The program ensures that the syntax is correct. The intent of this method is to generate a small number of requests which are thoughtfully devised in order to test specific features of MDBS. This program also assumes that one user will execute only one request at a time. The user is allowed the following options when using this test request generation package: generating a list of requests for later use, retrieving a list of requests to be executed in any order, modifying an existing list, or executing a list of requests.

2. Version B

Version B is a follow-on package to Version A. It therefore possesses all of the features contained in Version A. It should be noted that Version B adds the ability to use the concept of transactions. Recall that a transaction is a group of one or more requests. Thus the requirement of executing only one request at a time is removed.

D. THE SELECTION OF THE TEST REQUEST GENERATION PACKAGE

Because Version B contains all the features of Version A, Version B was selected as the test request generation package. Because this version arrived at the current implementation site of MDBS rather late in the review of performance evaluation tools, many of the desired features must be left for future development. This does not detract from the usefulness of the test request generation package as it stands.

E. THE UPGRADING PROCESS

The majority of the upgrading accomplished on the test request generation package consisted of ensuring that the syntax discrepancies due to compiler differences were removed. A reorganization of the file location of MDBS resulted in many changes to the programs.

F. FUTURE IMPROVEMENTS

Several enhancements to the request generation package may be desirable. Three major enhancements include the following: program generation of requests, simulation of mutltiple concurrent users, and development of a storage information package to aid in request selection.

1. <u>Program Generation of Requests</u>

In order to test MDBS, the test request generation package could be modified to contain a routine which generates random requests. The input to such a routine would include parameters such as the percentage of each type of request to be generated and the the query complexity. Query complexity involves changing the number of predicates in the requests. This ability would allow the evaluation team to easily determine which type of request is most efficient under MDBS.

2. <u>Simulation of Multiple Concurrent Users</u>

In order to evaluate the effect of concurrency control, MDBS must be tested while several users are using the system. By providing a way to link a user to the requests which are generated, the test request generation package would simulate mutiple users. This would avoid processing several separate files of requests. This would also result in repeatable experiments, in that the conditions resulting from executing the concurrent user requests could be duplicated.

3. <u>The Storage Information Package</u>

The storage information package would allow the experimenter to ask specific questions about the database storage information so that intelligent queries can be derived. The questions an experimenter might ask would include: What descriptors are associated with a certain attribute? What descriptor ids define a certain cluster number? or Where is cluster one stored?

This package could be implemented by sending messages to the backends. Each message would be associated with a routine which walks through the directory management tables and finds the appropriate information and sends it back to the contrcller. By evaluating the responses to the messages, more meaningful requests can be constructed in order to evaluate specific features of MDBS.

VI. ANALYSIS OF PERFORMANCE EVALUATION TOOLS

In Chapter I, we discussed the study phase of creating the tools. In Chapter II, we discussed the design phase. The development phase was outlined in Chapters III, IV, and V. In this chapter, we discuss the operational phase. This taxonomy of phases is outlined in detail in [Ref. 8]. More specifically, in this chapter, we discuss the performance evaluation tools with respect to several software engineering principlas.

A. BASIS OF ANALYSIS

In this section, we discuss the standards by which the evaluation tools are to be analyzed. The two major categories of the analysis are the ability to meet the objectives stated in the design phase and the ability to meet software goals. The standards are described in detail in [Ref. 9] and [Ref. 10].

The ability to meet objectives means that the tool possesses the capabilities outlined in the design phase. These capabilities were discussed in detail in Chapter II.

The performance evaluation tools will be evaluated also by their ability to meet five software goals. The first goal is that of modifiability. Modifiability includes the properties of extensibility, consistency, maintainability, and modularization. The second goal is that of reliability. Reliability includes the properties of possessing no blatent errors and of possessing error recoverability. The third goal is simplicity. This includes ease of use and singleness of purpose. Efficiency is the fourth goal. A tool will possess this goal if it contains no gross inefficiency.

The last software goal is that of understandability. Understandability means that the tool utilizes abstractions, modularity, and information hiding, and is supported with reasonable documentation.

B. ANALYSIS OF THE FILE GENERATION PACKAGE

The objectives of the file generation package were discussed in Chapter II. The objective that was not met by this tool is the ability to indicate whether values of the attributes are taken from random functions or predetermined sets of values. The random functions must be added at a future date.

The file generation package meets all goals with the exception of efficiency. Modifiability is achieved through the extensive use of modularization with respect to grouping like operations together. Reliability has been observed in that no errors have existed since the operational phase. Simplicity is demonstrated by using menu-driven operations in the file generation package. Lastly, understandability is achieved by religious use of abstraction of data and operation. The gross inefficiency in the package results from the use of a large array which is used to store the unique records which are generated. When a large number of records are to be inserted at one time, the time to compare the new record against all previously generated records is great. This concludes the evaluation of the test file generation package.

C. ANALYSIS OF THE DATABASE LOAD SUBSYSTEM

The objectives of the database load subsystem were discussed in Chapter II. The objective that was not met by this tool is the ability to vary the data placement strategy. This ability must be added at a future date.

The database load subsystem meets all goals with the exception of efficiency. Modifiability is achieved through the extensive use of modularization with respect to grouping like operations together. For instance, all of the routines to pass messages are grouped in send and receive modules which are kept in separate files. Reliability has been observed in that no errors have existed since the operational phase. Simplicity is demonstrated by using menu-driven operations. Lastly, understandability is achieved by religious use of abstraction both in the data and the operations. The gross inefficiency in the package results from the use of a large number of insert requests which are sent one at a time to the backends. This inefficiency could be reduced by grouping several insert requests into a transaction and then sending the transaction to the backends. It is also possible to save all type-C descriptors in the database load subsystem and send all of them to Insert Information Generation at the end of the directory table loading. This concludes the evaluation of the database load subsystem.

D. ANALYSIS OF THE REQUEST GENERATION PACKAGE

The objectives of the test request generation package were discussed in Chapter II. The objectives that were not met by this tool are the following enhancements: program generation of requests, simulation of multiple concurrent users, and development of a storage information package to aid in request selection. These abilities must be added at a future date.

The test request generation package meets all goals with the exception of possessing consistency. Modifiability is achieved through the extensive use of modularization with respect to grouping like operations together. For instance, all of the routines which are involved with creating a request are divided into modules each of which handles a distinct aspect of the request. This goal is seen throughout MDBS. Reliability has been observed in that no errors have existed since the operational phase. Simplicity is demonstrated by using menu-driven operations. Lastly, understandability is achieved by religious use of abstraction both in the data and the operations. Consistency may be achieved by altering the test request generation to use information stored in the files generated by both the test file generation package and the database load subsytem. These files could be used for the extraction of necessary information instead of prompting the user to re-enter data supplied earlier. It is the weakest link in establishing a tight performance evaluation environment. This is further discussed in the next section. This concludes the evaluation of the database load subsystem.

E. FUTURE DEVELOPMENTS

The most important future development should be the integration of the performance evaluation tools into a performance evaluation environment. In this way, the property of consistency of the tools will be attained. That is, the output of one tool can be used as input to the next tool in the logical sequence of the performance evaluation effort. This has been achieved in the test file generation package-database load subsytem interface. The next step would be to develop consistency between the database load subsystem-test request generation package interface.

This concludes the discussion on the analysis of the performance evaluation tools.

VII. CONCLUSIONS

In this thesis, we have discussed the development of the necessary tools for the performance evaluation of a multi-backend database system, known as MDBS. The basic motivation of the mutlti-backend database system (MDBS) was to develop an architecture which spreads the work of the database system among multiple backends. It was a major aim of this system to allow capacity growth by the use of additional disk drives and performance improvement by the use of additional backends. However, to verify the design and implementation, it is necessary to test the capability of MDBS in capacity growth and performance gain.

Three tools for the performance and capacity tests were investigated. The first tool was the file generation package which creates test files for any artificial database. The second tool was the database load subsystem which loads the artificial database into MDBS. The third tool was the request generation package. This package created test requests to query MDBS.

The following methodology was used to create an effective tool. First, the properties of an ideal tool were described. Then available existing programs were reviewed and evaluated to determine which program best meets the desired features. The programs were upgraded to ensure that they were compatible with the current implementation, and met the desired features. Lastly, the tools were analyzed with respect to meeting the desired properties and satisfying several software engineering goals.

The main goal was to develop the necessary tools to generate tests in measuring the extensibility of MDBS, i.e., how does MDBS perform as more backends are added?

Performance was expected to improve (maintain) as the number (size) of the backends (database) was increased. We feel that the tools developed herein will allow an easy and efficient means of measuring the extensibility of MDBS.

APPENDIX A

DESIGN SPECIFICATION OF THE TEST FILE GENERATION PACKAGE

This appendix contains the design of the test file generation package which is a subset of the shortened data-base load subsystem. The design consists of C language code for the function headings and their corresponding declarations. The body of the functions are given in English text.

```
/**********************/
/*   TEST      FILE  */
/*    GENERATION     */
/*     PACKAGE       */
/*      DESIGN       */
/**********************/


main_program()
begin
    generate(); /*generate the records*/
end



generate()
 /* This routine                                                   */
 /*       - generates a record template                            */
 /*       - generates/modifies sets of values for attributes       */
 /*       - generates descriptors                                  */
 /*       - generates records using the sets                       */

begin
    while ( TRUE )
    begin

          /*Ask the user for type of operation to be performed*/
          /*Take appropriate action*/
                    /* generate record template */
                    gen_tmpl();
                    /* generate descriptors */
                    gen_desc();
                    /* generate/modify sets */
                    gm_set();
                    /* generate the records */
                    gen_rec();
                    /* load the records */
                    db_load();
                    /*do nothing*/
    endwhile;
end
```

```
gen_tmpl()
   /* This routine generates a record template */
begin
   char  tfn(MFNLength + 1);     /* template-file name */
   char  c, dbid(DBIDLNTH+1), hold(MAX_FIELDS+1), temtyp;
   int   i, k, no_attr;
   FILE  *fopen(), *tmpl_fp;

   /* Get name of template file */
   /* Open template file */
   /* Get database ID from the template file*/
   /* Write database ID to template file */
   /* Get number of attributes */
   /* Write number of attributes to template file */
   /* Get attributes and value types */
   for (each attribute)
   begin
       /* Enter the attribute name*/
       /* Enter the value type: (s=string, i=integer)*/
   end    /* end for */
   /* Close template file */
end  /* end gen_tmpl */



gen_desc()

begin
   char  tfn(MFNLength + 1); /* template-file name */
   char  dfn(MFNLength + 1); /* descriptor-file name */
   char attr_name(ANLength),
     answer(5), desc_type, val_type, c, hold(3);

   int i, j, nc_attr;

   FILE *fopen(), *tmpl_fp, *desc_fp;

   /* Get the template-file name */
   /* Open template file */
   /* Get the name of the file for storing descriptors */
   /* Open descriptor file */
   /* Read thru Database ID to get */
        /* to number of attributes      */
   /* Get number of attributes */
   /* For each attribute get its descriptors (if applicable)*/
   for (each attribute)
   begin
       /* Read attribute */
       /* Get attribute name */
       /* Get value type for the attribute */
       /* Ask if attribute
             is to be a directory attribute*/
       if ( answer= yes )
       begin
          /* Write attribute name to descriptor file */
       /* Get descriptor type for attribute */
       /* Write descriptor type to descriptor file */
       if ( desc_type == 'C' | desc_type == 'c')
          gen_C(val_type,desc_fp);
       else
          gen_notC(val_type,desc_fp);
       /* Write end_of_data symbol to descriptor file */
       end
   end /* end for */
   /* Write end_of_file symbol to descriptor file */
```

```
    /* Close files */
end/*gen_desc*/



gen_C(val_type,desc_fp)

   char val_type;
   FILE *desc_fp;
begin
   char lowerb(AVLength), upperb(AVLength), hold(3);
   int fault,k;

   /* Get upper bounds for type 'C' descriptors */
   while ( TRUE )
   begin
      /* Get upper bound */
      if ( end of data)
           return;
         else
         begin
           /* Verify upper bound entry against */
         /* attribute value type             */
         /* Write NOBOUND and upper bound */
           /* to descriptor file             */
         end
     end
 end /* end gen_C  */



gen_notC(val_type,desc_fp)

   char  val_type;
   FILE *desc_fp;
begin
   char lowerb(AVLength), upperb(AVLength), hold(3);
   int fault, k;

   /* Get lower and upper bounds for descriptor */
   while ( TRUE )
   begin
      /* Get lower bound */
         if ( end of data)
           return;
         else
         begin
              /* Verify lower bound entry against */
           /* attribute value type             */
           /* Write lower bound to descriptor file */
         end
      /* Get upper bound */
      /* Verify upper bound entry against */
      /* attribute value type             */
      /* Write upper bound to descriptor file */
   end /* end while */
end/* end gen_notC */



gm_set()
   /* This routine generates/modifies sets of values. */
begin
   char tfn(MFNLength + 1); /* template-file name */
```

```
    char attr_name(ANLength + 1), answer, c, val_type,
         hold(AVLength +1);
    char tmptyp;
    int no_attr, k, i;

    FILE *fopen(), *tmpl_fp;


    /* get the template-file name */
    /* Open template file */
    /* Get number of attributes */
    for ( each attribute )
    begin
        /* Get attribute name */
        /* Get value type */
        /* Choose the action to be taken on attribute
                  n)  -  generate a new set for it
                  m)  -  modify an existing set for it
                  s)  -  do nothing with it                  */
        switch( answer )
        begin
           case 'n':
                 /* generate new set */
                 gen_set( val_type );
                 break;
           case 'm':
                 mod_set(val_type);
                 break;
           case 's':
                 break;
        end   /* end switch */
    end   /* end for */
    /* Close template file */
end   /* end gm_set */



gen_set(val_type)

    /* This routine generates a set */
    /*of values for an attribute. */

    char  val_type;
begin
    struct definition
    begin
         char elem(SetSize)(AVLength + 1);
         /* array for holding set elements */
         int no_elem;
         /* number of elements in set       */
    end set;

    char filnam(MFNLength + 1),
         entry(AVLength + 1), answer(5);
    int k, fault, limit;

    FILE *fopen(), *tmpl_fp;

    /* Get name of set file */
    /* Open set file */
    /* Accept elements for the set */
    while ( set is not full)
    begin
          /* Enter a value for the set */
          /* Verify set entry against attribute type */
          /* Check for set element duplication */
    end
    if ( set is full)
          /* Tell user*/
    /* Write set elements to set file */
    /* Write end_of_file symbol to set file */
    /* Close set file */
    /* Ask if user wants to modify it */
    if ( answer= yes )
       mod_set(val_type);
end /* end gen_set */




mod_set(val_type)
    /* This routine modifies a set */
    /* of values for an attribute. */

    char  val_type;
begin
    char    ofn(MFNLength + 1), /* old-file name */
            nfn(MFNLength + 1), /* new-file name */
            filnam(MFNLength + 1);

    char c, answer(5), entry(AVLength + 1), index(5);
    int i, k, fault,j;

    struct
      begin
        int   no_elem; /* number of elements in the set */
        char  rem_flag(SetSize); /* element removed flag */
        char  elem(SetSize) (AVLength + 1); /* elements */
      end set;

    FILE *fopen(), *set_fp;

    /* Get the name of the set to be modified */
    /* Open file */
    /* Read given file into array for manipulation */
    while ( TRUE )
    begin
       /* Ask what do you want to perform next?*/
               (p)  -  print the set elements and their indices
               (a)  -  add some elements to the set
               (r)  -  remove some elements from the set
                   (n)  -  nothing; done
       if ( answer = 'p' )
       begin
          /* Print elements of file */
       end/* end ( answer = 'p' ) */
       else if ( answer = 'a' )
       begin
           /* Add some elements */
           /* Check for set element duplication */
           /* Verify entry against */
           /* attribute value type */
           /* Add element to array, if correct*/
       end    /* end ( answer = 'a' ) */
       else if ( answer = 'r' )
       begin
          /* Remove some elements */
          /* Mark set elements for removal */
          /* Re-order array to reflect deletions */
       end  /* end ( answer = 'r' ) */
       else
          /* Nothing; done */
          break;   /* exit while */
    end  /* end while ( TRUE ) */
    /* Ask if user wants to store the modified set back
                     into the original file */
    /* Write array back into file designated*/
    /* Write end_of_file symbol to set file */
    /* Close set file */
end/* end mod_set */



gen_rec()
    /* This routine generates records using sets. */
begin
    char c;
    char hold(AVLength + 1);
    char attr_name(AVLength + 1);

    char  dbid(DBIDLNTH + 1),
          gr_records(MAX_RECORDS)(MRLength + 1);
    char  tfn(MFNLength + 1), /* template-file name */
          rfn(MFNLength + 1), /* record-file name */
       vfn(MFNLength + 1); /* temporary file name */

    struct
    begin
      int no_elem(MAX_FIELDS);
      char elems(MAX_FIELDS)(SetSize)(AVLength + 1);
    end values;

    FILE *fopen(), *tmpl_fp, *rec_fp, *stor_fp;

    int no_attr, k, i, j, count, gr_no_rec, max,
        rec_cnt, prod, index, old;

    /* Get the template-file name */
    /* Open template file */
    /* Get file for record storage */
    /* Open record file */
    /* Read database ID */
    /* Write database ID to storage file */
    /* Read number of attributes in a record */
    /* Read elements of files corresponding to */
    /* each attribute into an array           */
    for ( each attribute)
    begin
       /* Read the attribute name */
       /* Get the file name for the given attribute */
       /* Open file */
       /* Read elements of set into array */
       /* Close file */
    end   /* end for */
    /* Close template file */
    /* Calculate total possible number of unique records */
    /* Get the number of records to be generated */
    /* Determine feasibility of requested number */
    /* Generate records by choosing (at random) */
    /* a member from each of the given sets     */
    for ( each record)
    begin
        for (for each attribute)
        begin
          /* Get a value randomly from the set*/
        end
        /* Give some feedback to user of generation effort*/
        /* Check generated record for possible duplication */
    end
    /* Write generated records to file */
    /* Write end of file symbol to file */
    /* Let user know when completed*/
```

```
   /* Close file */
end/* end gen_rec */


int gr_isdigit(c)

   /* This routine determines whether a given */
   /*                character is a digit   */

   char c;
begin
   if ( c is a digit )
      return(TRUE);
   else
      return(FALSE);
end



gs_rand(num)

   /* This routine generates a random number */

   int num;
begin
     static long seed;
     static int temp;
     seed = seed * 24298 + temp + time(0);
     seed = seedmod199017;
     seed = (69069 * seed + 1);
     temp = (seed >> 8) & 32767;
     if ( num == 0 )
        return(temp);
     else
        return(temp mod num);

end
```

APPENDIX B

DESIGN SPECIFICATION OF THE DATABASE LOAD SUBSYSTEM

This appendix contains the design of the shortened database load subsystem. The design consists of C language code for the function headings and their corresponding declarations. The body of the functions are given in English text.

```
                    /*****************/
                    /*               */
                    /* Database Load */
                    /*    Design     */
                    /*               */
                    /*****************/

struct rtemp_definition template;

db_load()
/* This routine loads the directory tables and the database */
/* records.                                                 */
begin
          /* Initialize counters*/
          /* load the directory tables */
          dbl_dir_tbls();
          /* load the database records */
          dbl_records();
end


dbl_dir_tbls()
  /* This routine loads the directory tables. */
begin
   char    dbid(DBIDLNTH + 1),
           attrname(ANLength + 1),
           tfn(MFNLength + 1), /* template-file name */
           dfn(MFNLength + 1), /* descriptor-file name */
           valtype,
           str(),
           attrstr(DIL_AttrId +1),
           desctype;

   int at_id_no, desc_id_no;

   struct  desc_definition  descriptor;

   int   i, k, c;

   FILE *fopen(), *fptr;

   /* Initialize the database mailbox*/
   /* Get the name of the file containing */
   /* the template information */
   /* Read the database id */
```

```
    /* Read number of entries in the template, i.e.,*/
    /* number of attributes in a record            */
    /* Read the attribute names and the value types */
    /* and place the data in the template record    */
    for ( each attribute to be put in template)
    begin
        /* Read an attribute */
        /* Read the corresponding value type */
    end
    /* Create attribute table for the database in backends */
    DBL_S$Create(dbid);
    /* Get the name of the file containing the descriptors */
    /* Read the directory attributes and their */
    /* corresponding descriptors               */
    /* Initialize the attribute counter */
    while ( not the end of data )
    begin
         /* Read an attribute */
         /* Read corresponding descriptor type A,B, or C */
         /* Add the attribute name to the attribute table */
         DBL_S$Atm_insert(dbid,attrname,&desctype);
         if (desctype == 'c' | desctype == 'C')
              /* Send the attribute to IIG */
              DBL_S$send_typeC(dbid,attrname,at_id_no);
         /* Using the template, find the value */
         /* type for the attribute */
         /* Read the corresponding descriptors*/
         /* for the attribute */
         /* Inititialize the descriptor id */
         while ( More descriptors )
         begin
              /* Get lower bound */
              /* Get upper bound */
              /* Add the descriptor to DDIT */
              DBL_S$Desc_add(dbid,attrname,&desctype,
                 &descriptor,&valtype,at_id_no,desc_id_no);
              /* Increment the descriptor id count */
         end /* end while */
         if ( desctype != C )
         begin
            /* Add the catchall descriptor to DDIT */
            DBL_S$Catchall(dbid,attrname,
               &desctype,at_id_no,desc_id_no);
         end /* end if */
         /* Increment the attribute count */
      end /* end while */
      /* Close descriptor file */
end/* end dbl_dir_tbls */



dbl_records()
begin
    char    dbid(DBIDLNTH + 1),
            rfn(MFNLength + 1), /* record-file name */
            req(REQLength),
            record(80);
    struct  rtemp_definition   *tmpl_ptr,
                               *get_tmpl_ptr();
    int i, c;

    FILE *fopen(), *fptr;

    /* Get the name of the file */
    /* containing the records to be loaded */
    /* Read the database id */
    /* Get the record template for the database */
    while ( more records exist )
    begin
          /* While there are more records */
          /* Read the next one            */
          /* Construct a request to insert record */
          dbl_construct_ins( tmpl_ptr, record, req );
          /* Send the request to Request-Preparation */
          DBL_S$TrafUnit(dbid, req);
      end
   end /* end dbl_records */



dbl_construct_ins(tmpl_ptr, record, req)

    struct  rtemp_definition  *tmpl_ptr;
    char    req(), record();
begin
    int i, j, k, p, entry_no;

    /* Load the initial part of request */
    while ( not the end of the record )
    begin
         /* Load the attribute name */
         /* Load the attribute value */
    end
    /* Load the end of request */
end
```

APPENDIX C

DESIGN SPECIFICATION OF THE TEST REQUEST GENERATION PACKAGE

The program specification for the test request generation and execution package is shown in this appendix. This design is the result of the work of Dr. Kerr, who headed the design of the original test request generation package.

The Top Level of Test Request Generation Package

This program can be used to test and demonstrate MDBS. The execution of this program is called a session. Each session can be divided into any number of subsessions. During a subsession the user can do one of the following:

(A) Execute a list of requests that was previously stored in a file.

(B) Prompt the user for a list of requests to be stored in a file for later use.

(C) Retrieve a list of requests that were previously stored in a file and then allow the user to select requests from that list for execution. This selection can be done in any order. The user will also be able to enter a new request to be executed.

(D) Modify an existing list of requests that was previously stored in a file.

In this version, requests are allowed to be grouped as transactions. A request is sent to MDBS. The program waits for a response before sending the next request or will continue to execute without response if the user so desires.

Output may be directed to the user's terminal or to a file or to both.

Program Specifications

```
                    /*********************/
                    /*                   */
                    /* Test    Request   */
                    /*    Generation     */
                    /* Package Design    */
                    /*                   */
                    /*********************/
task   MDBS Test;
scalar   more-subsessions; /* flag: TRUE - continue,
                                  FALSE - stop */

  Print initial message to user;
  more-subsessions := TRUE;
    while   more-subsessions    do
        perform   SUBSESSION;
      Prompt for continue message;
      Read continue message;
        if   user does not want to continue
           then
             more-subsessions := FALSE;
        end if
    end      while   ;
end      task   ;

procedure   SUBSESSION;

  /* During a subsession the user is able                      */
  /*          to generate a group of requests. (NEW__LIST)     */
  /*          to modify an old list of requests. (MODIFY)      */
  /*          to select requests, one at a time from a list    */
  /*             of requests. (SELECT)                         */
  /*          to run a group of requests. (OLD__LIST)          */

scalar   current-request-file; /* The name of the file */
     /* Initial value should be NULL. This name must be        */
      /*  retained from one subsession to the next.            */
scalar   type-of-subsession; /* Possible values are NEW__LIST,
      MODIFY, SELECT and OLD__LIST */

  Prompt for next type-of-subsession;
  Read next type-of-subsession;
    case   type-of-subsession    value
      NEW__LIST: /* Enter a new request-list */
           perform   NEW__LIST__SUB( current-request-file) ;
      MODIFY: /* Modify an old list */
           perform   MODIFY__SUB( current-request-file );
      SELECT: /* Select requests, one at a time, from an */
              /* existing request-list */
           perform   SELECT__SUB( current-request-file );
      OLD__LIST: /* Execute an existing request-list */
           perform   OLD__LIST__SUB( current-request-file) ;
        otherwise  : Print error message;
    end      case  ;
end      procedure  ;


procedure   NEW__LIST__SUB(   output  : current-request-file );
    scalar   current-request-file; /* name of the file */

  /* Asks user for requests - one at a time.                      */
  /* Saves list of requests in a file with file-name given by */
  /* user.                                                        */

scalar   request-list-file-name;
       /* of file to use to store the requests */
record   request;
scalar   next-step;
```

```
        /* I(nsert), R(etrieve), U(pdate), D(elete) or F(inish) */

    Prompt for request-list-file-name;
    Read request-list-file-name;
    Open file( request-list-file-name ) output;
    perform  ENTER__AND__SAVE__REQUESTS( request-list-file-name );
    Close file( request-list-file-name );
    current-request-file := request-list-file-name;
  end     procedure  ;


  procedure MODIFY__SUB(   input/output  : current-request-file );
      scalar   current-request-file; /* The name of the file */

    /* Retrieve an old request-list and then allow the user to   */
    /* modify it.  Requests are examined one at a time allowing  */
    /* changes to be made to each request in turn.  A change     */
    /* can be                                                    */
    /*      add new request before this one.                     */
    /*      modify this request.                                 */
    /*      remove this request.                                 */
    /*      make no changes to this request.                     */
    /* Note that we must have a way to append new requests at    */
    /* the end of the input request list.                        */
    /*                                                           */
    /* The input file ( called input-request-file ) may be       */
    /* either the current-request-file or a different existing   */
    /* request file.                                             */
    /*                                                           */
    /* The output file ( called new-request-file ) may be        */
    /* either the next version of the input-request-file or a    */
    /* new file.                                                 */

  scalar   input-request-file; /* The list of requests
                                   to be modified. */
  scalar   new-request-file; /* The new list of requests. */
  scalar   next-version; /* flag:TRUE-set new-request-file to */
    /*next version of input-request-file, FALSE-get new name*/
  record   request;
  scalar   more-requests-in-input-request-file;/*continue flag*/
  scalar   more-requests-to-enter; /* continuation flag */
  scalar   change-type; /* ADD, MODIFY, REMOVE, or NOCHANGE    */

  scalar   next-step;
        /* I(nsert), R(etrieve), U(pdate), D(elete) or F(inish)*/

    /* Determine input-request-file to be modified. */
      perform   DETERMINE__INPUT__FILE( current-request-file,
                             input-request-file );
    open file( input-request-file ) input;

  /*Determine if user wants the name of the new-request-file*/
  /* to be the next version of the input-request-file*/
  /* or a new name.*/
    Prompt user to determine next-version;
    Read next-version;
      if   next-version
          then
            Set new-request-file to next version of
                       input-request-file;
          else
            Prompt for new-request-file name;
            Read name of new-request-file;
      end     if   ;
    open file( new-request-file ) output;

    Read first request from input-request-file;
    more-requests-in-input-request-file := TRUE;
    while   more-requests-in-input-request-file   do
      Prompt user for change-type for this request;
      Read change-type;
        case   change-type   value
          ADD: /* enter and save the next request */
              perform   GET__NEW__REQUEST( request );
              Write request intc new-request-file;
          MODIFY:
              Prompt and get modified request from user;
              Write new request into new-request-file;
              Read next request from input-request-file;
          REMOVE:
              Read next request from input-request-file;
          NO_CHANGE:
              Write current request into new-request-file;
              Read next request from input-request-file;
            otherwise  : Print system error message;
        end      case  ;
    end      while  ;

  /* Note that at this point all the cld requests have been */
  /* processed. However it is possible that the user wants */
  /* to append more requests.                              */

  Prompt user that input file has been processed, but that
      more requests may still be appended;
    perform   ENTER__AND__SAVE__REQUESTS(new-request-file);
  close file( input-request-file );
  close file( new-request-file );
  current-request-file := new-request-file;
end     procedure  ;


procedure   SELECT__SUB(input/cutput  : current-request-file );
    scalar   current-request-file; /* The name of the file */

  /* Retrieve an old list of requests.                  */
  /* Allow user to select from this list.               */
  /* Also allow user to enter new request.              */

scalar  input-request-file; /* file containing requests*/
array   requests( MAX__NUMBER__CF__REQUESTS );
               /* frcm input-request-file */
    scalar   number-of-requests; /* The actual number in */
      /* input-request-file must be less than          */
      /* MAX__NUMBER__CF__REQUESTS                     */
    scalar   request-number; /* of the request chosen */
record   new-request; /* Provided by user. */
record   response; /* to the request being executed. */

scalar   more-to-execute; /* flag to control loop */
scalar   next-operation;
      /* Values can be REQUEST__NUMBER, DISPLAY,*/
      /* NEW__REQUEST or STOP                   */

  /* Determine the new input-request-file to use for */
  /* this subsession. */
    perform   DETERMINE__INPUT__FILE( current-request-file,
                       input-request-file );
  open( input-request-file );
  Read and store input-request-file into requests checking that
    number-of-requests is less than MAX__NUMBER__OF__REQUESTS;
  close( input-request-file );
    perform   DISPLAY( requests );

 /*Determine whether response is to go to CRT, file or both*/
    perform   OUTRSPOEMAT;
```

```
more-to-execute := TRUE;

  while   more-to-execute   do
    Prompt user for next-operation /*should be either*/
    /* request-number, a request-to-display or a     */
    /* new-request                                   */
    Read next-operation;
      case   next-operation   value
        REQUEST__NUMBER:
            Check that request-number is less than
                     number-of-requests;
              perform   EXECUTE( requests(request-number),
                     response );
            /* Output the response to CRT, file or CRT_&file,
                     as appropriate. */
              perform   OUTM$RESPONSE( response );

        DISPLAY:   perform   DISPLAY( requests );
        NEW__REQUEST:
              perform   GET__NEW__REQUEST( new-request );
              perform   EXECUTE( new-request, response );
         /* Output the response to CRT, file or CRT_&file,
                        as appropriate. */
              perform   OUTM$RESPONSE( response );

        STOP: more-to-execute := FALSE;
          otherwise  : print error message;
      end       case  ;
  end       while  ;

  perform   OUTM$FINISH;
current-request-file := input-request-file;

end      procedure  ;


procedure   OLD__LIST__SUB( current-request-file );
  scalar   current-request-file; /* The name of the file */

  /* Retrieve and execute an old list of requests. */

scalar   input-request-file /* The file containing requests*/
record   request;
record   response; /* to a request that has been executed. */

  /* Determine the new current-request-file to use for this*/
  /* subsession. */
  perform   DETERMINE__INPUT__FILE( current-request-file,
                   input-request-file );
  Open( input-request-file ) input;
  Read first request from input-request-file;

  /* Determine whether response is to go to CRT, file or both. */
  perform   OUTM$FORMAT;
  while   more-requests   do
      perform   EXECUTE( request, response );
    /* Output the response to CRT, file or CRT_&file, as */
    /* appropriate. */
      perform   OUTM$RESPONSE( response );
    Read next request from input-request-file;
  end       while  ;

  perform   OUTM$FINISH;
  close( input-request-file );
  current-request-file := input-request-file;

end      procedure  ;
```

```
procedure   ENTER__AND__SAVE__REQUESTS
     (   input  : request-list-file-name );
scalar   request-list-file-name;
        /* of file to use to store the requests */
record   request;
scalar   next-step;
 /* I(nsert), R(etrieve), U(pdate), D(elete) or F(inish) */

   next-step := I;
     while   next-step ^= F   do
       Prompt for next-step;
         case   next-step   value
              I: /* enter and save the next insert request */
            perform   INSERT__SUB( request );
                    Write request into request-list-file-name ;
              R: /*enter and save next retrieve request */
                     perform   RETRIEVE__SUB( request );
                    Write request into request-list-file-name ;
              U: /* enter and save the next update request */
                     perform   DELETE__SUB( request );
                    Write request into request-list-file-name ;
              D: /* enter and save the next delete request */
                     perform   DELETE__SUB( request );
                    Write request into request-list-file-name ;
              F: /* Finish entering requests */
                      otherwise  : Print error message;
                 end     case   ;
             end     while   ;
end     procedure   ;


procedure   DETERMINE__INPUT__FILE(   input  :
                                      current-request-file,
                                      output  : input-request-file );
              scalar   current-request-file;
              scalar   input-request-file;

   /* Determine the input file to be used.  It may be either */
   /* the current-request-file or a different existing       */
   /* request file.                                           */

           scalar   modify-current-file-flag;
                          /* TRUE - select new input file */

              if    current-request-file is NULL
                 then
                   Prompt for name of input-request-file;
                   Read name of input-request-file;
          else   /* Determine if user wants to use the */
             /* current-request-file or a different old file. */
            Prompt user to determine modify-current-file-flag;
            Read modify-current-file-flag;
              if    modify-current-file-flag
                 then
                   Prompt for name of input-request-file;
                   Read name of input-request-file;
                 else
                   input-request-file := current-request-file;
              end      if  ;
       end if  ;
end procedure  ;


procedure   GET__NEW__REQUEST(   output  : request );
              record   request; /* to be obtained from user */
        /* Prompts user for information necessary to enter a */
        /* new request. Returns the request.                 */

            scalar   request-type;
             /* I(nsert), R(etrieve), U(pdate) or  D(elete) */

          Prompt fcr next request-type;
          Read request-type;
                case    request-type    value
                  I:    perform    INSERT__SUB( request );
                  U:    perform    UPDATE__SUB( request );
                  D:    perform    DELETE__SUB( request );
                  R:    perform    RETRIEVE__SUB( request );
                    otherwise   : Print error message;
                end       case   ;

end      procedure  ;


procedure    DISPLAY(    input   : requests );
          /* Display the requests and their numbers at the */
            /* terminal.                                    */

        array    requests( MAX__NUMBER__OF__REQUESTS );
                        /* to be displayed. */

end      procedure  ;


procedure    EXECUTE(    input   : request,
                                  output   : response );
   /* Ask MDBS to execute this request. Return the response. */

        record request; /* to be executed */
        record respcnse; /* to the execution of the request */

end      procedure  ;
```

LIST OF REFERENCES

1. Naval Postgraduate School Report NPS52-83-006, *The Design and Analysis of a Multi-backend Database System for Performance Improvement, Functionality Expansion and Capacity Growth (Part I)*, by Hsiao, David K., and Menon, Jaishankar, June, 1983.

2. Naval Postgraduate School Report NPS52-83-007, *The Design and Analysis of a Multi-backend Database System for Performance Improvement, Functionality Expansion and Capacity Growth (Part II)*, by Hsiao, David K., and Menon, Jaishankar, June, 1983.

3. Naval Postgraduate School Report NPS52-83-008, *The Implementation of a Multi-backend Database System (MDBS): Part I - Software Engineering Strategies and Efforts Towards a Prototype MDBS*, by Kerr, Douglas S., Orooji, Ali, Shi, Zong-Zhi, and Strawser, Paula, June, 1983.

4. Naval Postgraduate School Report NPS52-82-008, *The Implementation of a Multi-backend Database System (MDBS): Part II - The First Prototype MDBS and the Software Engineering Experience*, by Higashida, Xingui He, Hsiao, David K., Kerr, Douglas S., Orooji, Ali, Shi, Zong-Zhi, and Strawser, Paula, June, 1982.

5. Naval Postgraduate School Report NPS52-83-003, *The Implementation of a Multi-backend Database System (MDBS): Part III - The Message-Oriented Version with Concurrency Control and Secondary-Memory-Based Directory Management*, by Boyne, Richard D., Demurjian, Steven A., Hsiao, David K., Kerr, Douglas S., and Orooji, Ali, March, 1983.

6. Bogdanowicz, Robert, Crocker, Michael, Hsiao, David K., Ryder, Curtis, Stone, Vincent, and Strawser, Paula, *Experiments in Benchmarking Relational Database Machines*, M.S. Thesis, Naval Postgraduate School, Monterey, California, June, 1983.

7. Stone, Vincent C., *Design of Relational Database Benchmarks*, M.S. Thesis, Naval Postgraduate School, Monterey, California, June, 1983.

8. Gore, Marvin and Stube, John, *Elements of System Analysis*, Wm. C. Brown Publishing, 1983.

9. Ross, Douglas T. and others, "Software Engineering: Processes, Principles, and Goals", *Computer Magazine*, pp.54-56, May 1975.

10. Wasserman, Anthony T., _Tutorial: Software Engineering Environments_, pp.15-35, IEEE Computer Society, 1981.

BIBLIOGRAPHY

Hancock, Les and Krieger, Morris, The C Primer, McGraw-Hill Book Company, N.Y., 1983.

Kernnigan, Brian and Ritchie, Dennis M., The C Programming Language, Prentice-Hall, 1978.

PCL11-B Parallel Communication Link Differential, Digital Equipment Corporation, Maynard, Mass., 1979.

RSX-11M/M-PLUS Executive Reference Manual, AA-H265A-TC, Digital Equipment Corporation, Maynard, Mass., 1979.

VAX/VMS System Services Reference Manual, AA-D018B-TE, Digital Equipment Corporation, Maynard, Mass., 1980.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Dudley Knox Library, Code 0142 Naval Postgraduate School Monterey, California 93943	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
4.	LT Joseph G. Kovalchik 113 Roosevelt Street Edwardsville, Pennsylvania 18704	2
5.	Office of Research Administration Code 012A Naval Postgraduate School Monterey, California 93943	1
6.	Computer Technologies Curricular Office Code 37 Naval Postgraduate School Monterey, California 93943	1

8