NPS-52-82-008

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THE IMPLEMENTATION OF A MULTI-BACKEND DATABASE SYSTEM (MDBS): PART II - THE FIRST PROTOTYPE MDBS AND THE SOFTWARE ENGINEERING EXPERIENCE

Xingui He, Masanobu Higashida, David K. Hsiao,
Douglas S. Kerr, Ali Orooji, Zong-Zhi Shi,
and Paula Strawser

July 1982

Approved for public release; distribution unlimited

Prepared for: Naval Postgraduate School
Monterey, CA 93940

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral J. J. Ekelund
Superintendent

David A. Schrady
Acting Provost

The work reported herein was supported by Contract N00014-75-C-0573 from the Office of Naval Research.

This report was prepared by:

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS52-82-008	2. GOVT ACCESSION NO. AD-A119 161	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (*and Subtitle*) THE IMPLEMENTATION OF A MULTI-BACKEND DATABASE SYSTEM (MDBS): PART II - THE FIRST PROTOTYPE MDBS AND THE SOFTWARE ENGINEERING EXPERIENCE		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(*s*) Xingui He, Masanobu Higashida, David K. Hsiao, Douglas S. Kerr, Ali Orooji, Zong-Zhi Shi, Paula Strawser		8. CONTRACT OR GRANT NUMBER(*s*) N00014-75-C-0573
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 4115-A1
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940		12. REPORT DATE July 1982 13. NUMBER OF PAGES 149
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*)

backend database system, database system implementation, database computer, database machine, software engineering, database.

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

The Multi-Backend Database System, MDBS, uses one microcomputer as the master or controller, and a varying number of microcomputers as slaves or backends. No special hardware is required. The backends are configured in a parallel manner. A new backend may be added by replicating the existing software on an addition minicomputer.

A prototype MDBS is being implemented in order to carry out design verification and performance evaluation studies. This report is the second in a series which describe the MDBS implementation. First, the overall design and implementation of MDBS is given. Then, the controller and the backends functions are described in detail.

In order to facilitate performance evaluation experiments, a program to generate test data and a program to generate test requests are required. The former program was described in the first report. The latter program is described in this report.

Our goal is not limited to the production of a prototype MDBS, but is also aimed toward the application of software engineering techniques to the development of the system. Thus, the software engineering techniques being used are also discussed.

The appendicies contain the detailed designs for the controller subsystem, one of the two modules in the backends subsystem (the other module, the directory management, was contained in the first report) and the test request generation module.

PREFACE

This work was 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. 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) and consists of the staff, graduate students, undergraduate students, Visiting Scholars and faculty for conducting research in database systems. Dr. Douglas S. Kerr, Associate Professor of Computer and Information Science at the Ohio State University is the present Director of Laboratory.

Since July 1, 1982, Dr. Hsiao assumed the Chairmanship of the Computer Science Department at the Naval Postgraduate School and continued the funded research at the Naval Postgraduate School. The Laboratory for Database Systems Research will be moved to the Naval Postgraduate School (NPS) in June of 1983 and supported by DEC, ONR and NPS. This technical report was drafted at the Ohio State University and completed at the Naval Postgraduate School.

We would like to thank all those who have helped with the MDBS project. In particular, the MDBS design and analysis were developed by Jai Menon. (Now, Dr. Jai Menon of IBM Research Laboratory, San Jose.) He also provided much help in the detailed designs. Several graduate students are involved with MDBS project: Richard Boyne, Patti Dock, and James Kiper. Several undergraduate students are also involved with the project: Julie Bendig, Raymond Browder, Chris Jeschke, Drew Logan, Jim Mckenna, William Mielke, and Joe Stuber. Tamer Azsu and Steven Barth provided much help in the detailed designs and coding. Jose Alegria, Tom Bodnovich and David Brown contributed background material which was necessary for making our design decisions.

TABLE OF CONTENTS

LIST OF FIGURES

1.0 PROTOTYPING A MULTI-BACKEND DATABASE SYSTEM (MDBS)

Four approaches to the running of a database management system have been proposed in the literature:

(1) Running the database management system along with all other software on a single general-purpose computer, known as the host.

(2) Running the database management system on a second general-purpose computer system, known as the backend. This is known as the single-backend software approach.

(3) Developing a special-purpose database machine with specially designed hardware to perform the database management functions. This is known as the database machine or hardware backend approach.

(4) Running the database management system on multiple general-purpose computers. This is known as the multi-backend software approach.

Database management systems built using the first approach have some limitations, e.g., as the database grows and the rate of requests to database system increases, the host computer performance decreases. Database management systems built using the second approach have the same limitation, i.e., the performance of the single-backend system also decreases. Thus, overall performance of the host and backend will be degraded. The third approach may be promising, but not until the cost-effectiveness of this approach is demonstrated.

The fourth 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. This approach requires the development of an innovative software design which allows the addition of more backends of the same type and the replication of the software on the new backends without major system interruption. Thus, it does not require the development of any new hardware, but only the development of a new and replicable software architecture and a new and parallel hardware configuration. This report is the second in a series [Kerr82] which describes the development of a multi-backend database system known as MDBS as a prototype for experimenting with the fourth approach.

1.1 Logical Description of MDBS

In this section, we give a brief review of MDBS. Full details on the design and analysis of MDBS can be found in [Hsia81a] and [Hsia81b]. The first report in this series [Kerr82] gives a more detailed overview.

1.1.1 The MDBS Hardware and Software Organization

An overview of MDBS hardware organization is shown in Figure 1. The controller and the backends are connected by a broadcast bus. The controller will broadcast each request to all backends at the same time over this bus. Furthermore, there will be minimal broadcasting from one backend to the other backends.

Each backend is given a number of dedicated disk drives. The data from each file is distributed across all the backends. Each backend will then process the data from its own disk drives. Because each file is spread across all the backends, all backends can now execute the same request in parallel. Request execution at a backend is handled by having a queue of requests at the backend. When a backend finishes executing one request it can start executing the next request. In view of the execution mode, MDBS is a multiple-instruction-and-multiple-data-stream (MIMD) organizaton.

The data model chosen for the system is the attribute-based data model [Hsia70]. In MDBS the database consists of files of records. Each record is a collection of keywords, optionally followed by a record body. A keyword is made of an attribute-value pair such as <SALARY,12000> where $12,000 is the value of the attribute SALARY. A record body is a string of characters not used by MDBS for search purposes. An example of a record without a record body is shown below.

```
    ( <FILE,Employee>, <EMPLOYEE_NAME,Smith>, <CITY,Columbus>,
          <SALARY,12000>, <SERVICE,10> )
```

The first attribute-value pairs in all records of a file are the same. In particular, the attribute is FILE and the value is the file name. For example, the above record is from the Employee file. When dealing with the records of the same file, we frequently omit the first attribute-value pair,

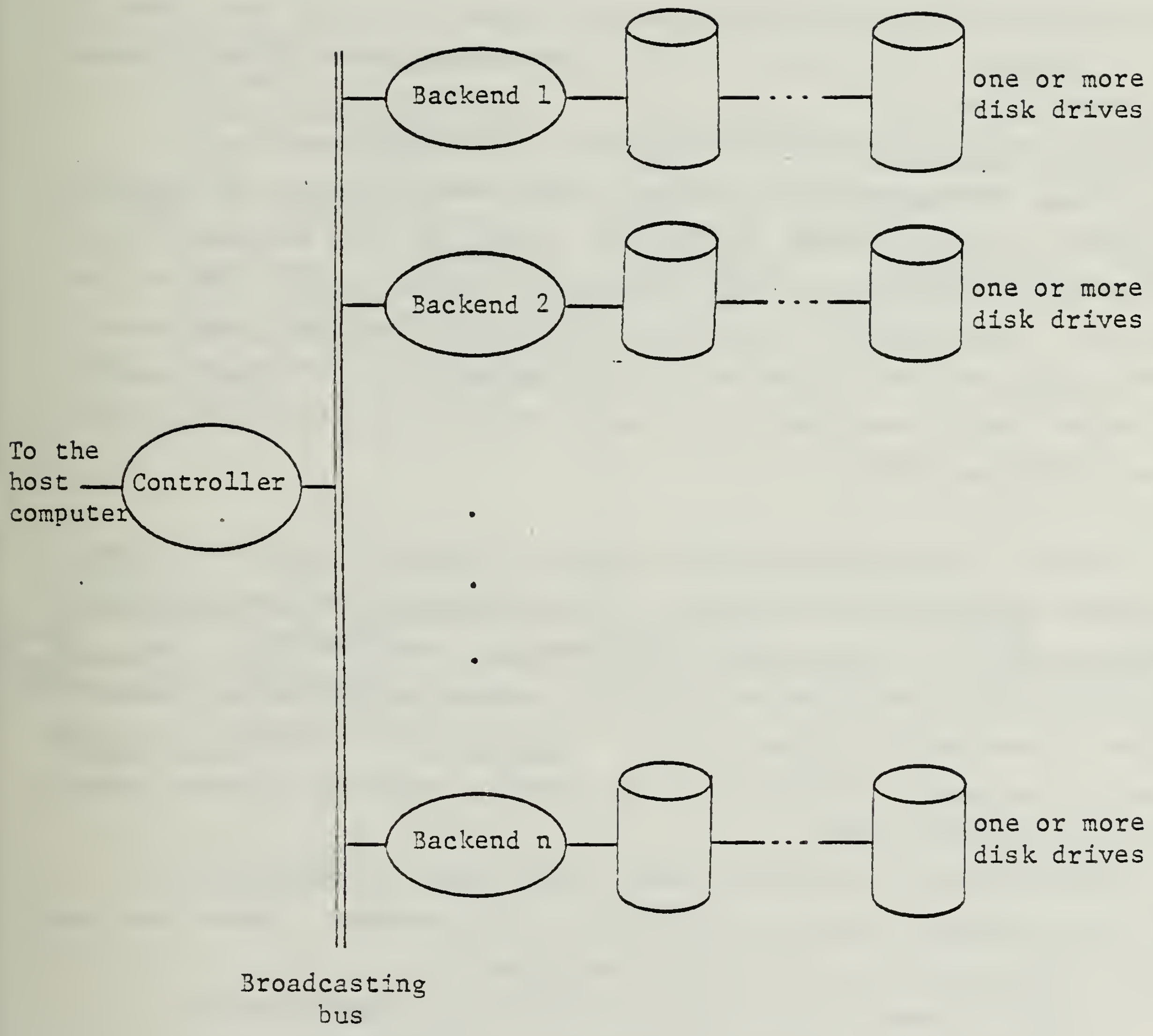


Figure 1. The MDBS Hardware Organization

i.e., the file name, for illustration.

For performance reasons, records are logically 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 employed in Columbus, making at least \$20,001 but not more than \$25,000 and with at least 11 but not more than 15 years of service. Thus records of this cluster are grouped by the following three descriptors:

(CITY=Columbus), (20001=<SALARY=<25000), (11=<SERVICE=<15).

MDBS performs its operations by clusters. Thus finding records of employees in Columbus making between \$21,000 and \$22,000 per year and with 12 to 13 years experience would require the retrieval of records in the cluster just described. Other retrieval requests such as to find records of employees in Columbus making between \$21,000 and \$28,000 and with 12 to 13 years experience might require additional retrieval of records from other clusters than the one identified above.

In order to allow efficient processing of requests, records in a cluster are spread across all the backends. Thus each backend needs to search only its portion of the cluster. Given a user request, there must be a way, of course, first to determine which clusters to search and then to determine the location of records in a given cluster. To perform this task, MDBS utilizes available descriptor information. For example, given the previous request for finding employees where

(CITY=Columbus) and (21000=<SALARY=<28000) and (12=<SERVICE=<13)

MDBS first determines that two clusters must be searched. These are the clusters identified by the two sets of descriptors:

{ (CITY=Columbus), (20001=<SALARY=<25000), (11=<SERVICE=<15) }
{ (CITY=Columbus), (25001=<SALARY=<30000), (11=<SERVICE=<15) }

After the clusters are identified, MDBS must then determine the disk addresses of the clusters at each backend. Finally MDBS will cause each backend to retrieve from its disks the records so addressed.

The execution phases of a retrieval request are summarized in Figure 2. Descriptor search determines the descriptors that correspond to the request.

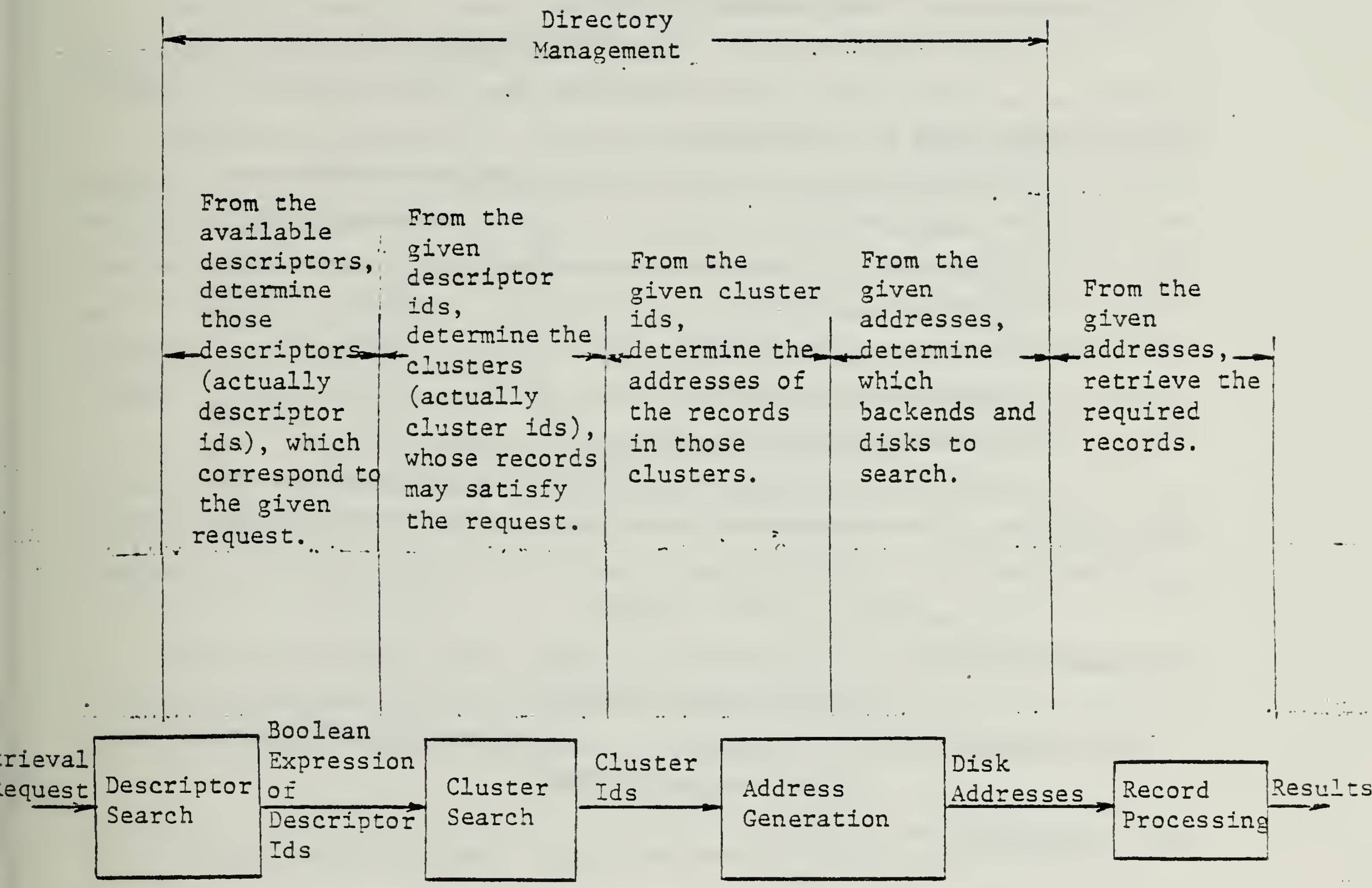

Directory Management
From the available descriptors, determine those descriptors (actually descriptor ids), which correspond to the given request.
From the given descriptor ids, determine the clusters (actually cluster ids), whose records may satisfy the request.
From the given cluster ids, determine the addresses of the records in those clusters.
From the given addresses, determine which backends and disks to search.
From the given addresses, retrieve the required records.
rieval Request
Descriptor Search
Boolean Expression of Descriptor Ids
Cluster Search
Cluster Ids
Address Generation
Disk Addresses
Record Processing
Results

Figure 2. Execution Phases of a Retrieval Request

In our example, there are four descriptors corresponding to the request; namely,

(CITY=Columbus), (20001=<SALARY=<25000),
(25001=<SALARY=<30000), (11=<SERVICE=<15).

In order to save space and to save processing time each descriptor is identified by a *descriptor id*. For example,

Descriptor	Descriptor Id
(CITY=Columbus)	D15
(20001=<SALARY=<25000)	D125
(25001=<SALARY=<30000)	D126
(11=<SERVICE=<15)	D250

Thus the output of the descriptor search phase is the Boolean expression of descriptor ids

D15 and (D125 or D126) and D250 (1)

corresponding to

$$\text{(CITY=Columbus) and} \left\{ \begin{array}{c} \text{(20001=<SALARY=<25000)} \\ \text{or} \\ \text{(25001=<SALARY=<30000)} \end{array} \right\} \text{and (11=<SERVICE=<15)}$$

which identifies two clusters.

The next phase, *cluster search*, must take the Boolean expression in (1) and actually determine the corresponding clusters. As with descriptors, clusters are also identified by ids, known as *cluster ids*, for example

Descriptor Ids	Cluster Id
D15, D125, D250	C17
D15, D126, D250	C22

The final two phases, see Figure 2, are *address generation* (to find the disk addresses, e.g., A3546 and A3547, corresponding to each cluster id, e.g., C17) and *record processing* (to retrieve the actual records so addressed and extract the fields required).

Descriptor search, cluster search and address generation together form the major portion of directory management.

Concurrent processing of requests is facilitated in MDBS. Executing one request at a time at a backend will frequently leave the backend's CPU idle while waiting for its disk to access records. Since the MDBS hardware organization provides multiple disk drives per backend, it is possible for a backend to support concurrent processing of requests from different users. However, a mechanism to control concurrent access to data must then be provided. Because all directory management is based on the concept of clusters, it is also logical to design a concurrency control mechanism based on clusters. Thus, the mechanism used in MDBS is centered on the concept of clusters. In particular, the concurrency control mechanism will lock clusters to prevent conflicting access to the same clustered data.

The general method used by MDBS in processing a retrieval request is described and summarized in Figure 3. In the next section, we will show how this processing is carried out among the controller and the backends.

1.1.2 Distribution of Request Execution Among Controller and Backends

In the previous section, we mentioned how the database was distributed across the backends. However, we did not discuss the placement of directory data and the distribution of the directory processing in directory management. In order to minimize the time for directory management and to facilitate record update, the directory data is duplicated at all backends. On the other hand, the directory processing is not duplicated at each backend. For instance, the descriptor search phase is divided among the backends. Each backend must find a different subset of descriptor ids. It then broadcasts its results to all the other backends.

In Figure 4, we summarize how directory management is performed at a backend. A retrieval request is received from the controller. Then the backend performs a descriptor search on its portion of the request and broad-

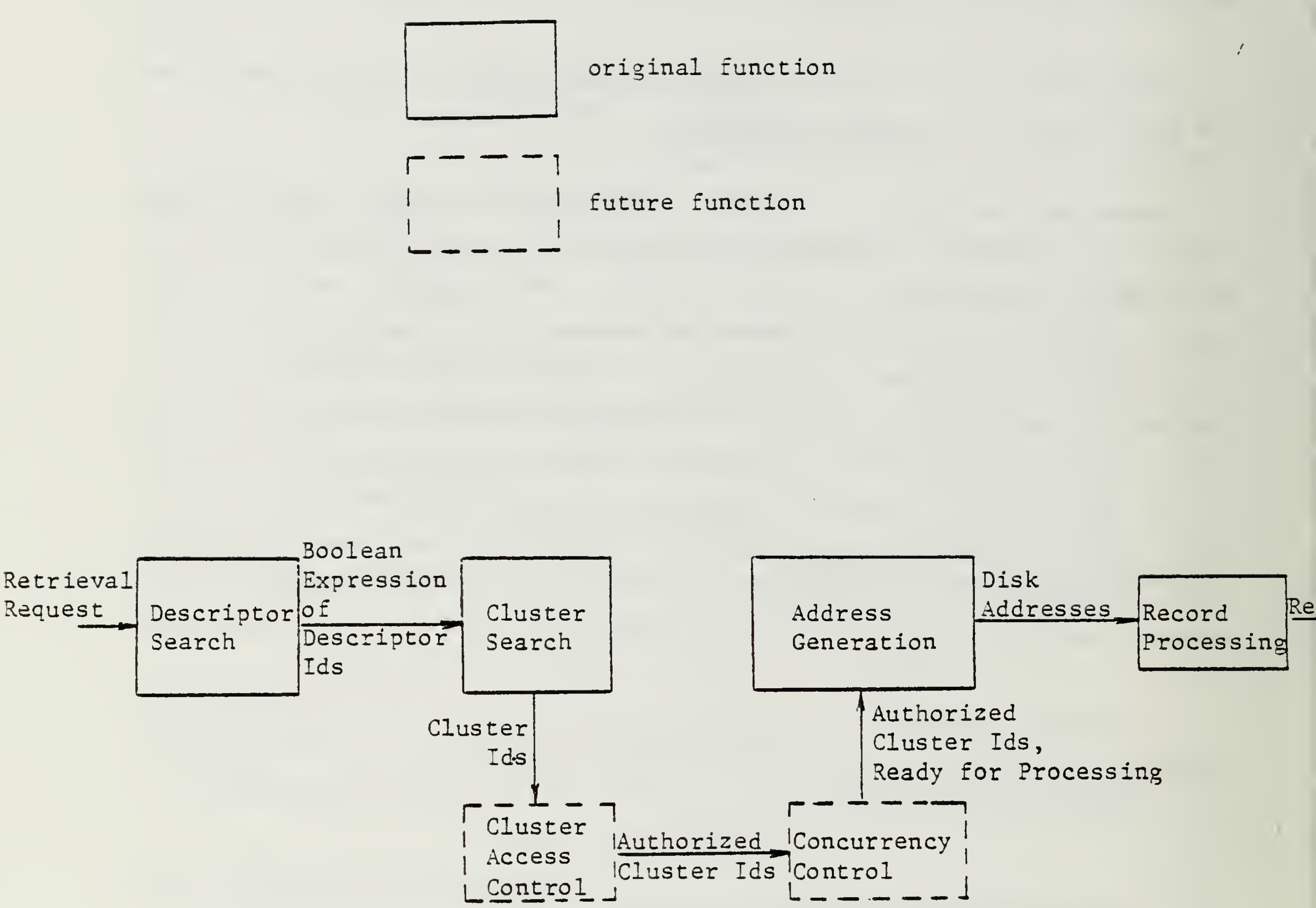


Figure 3. Execution of a Retrieval Request in the Presence of Access Control and Concurrency Control

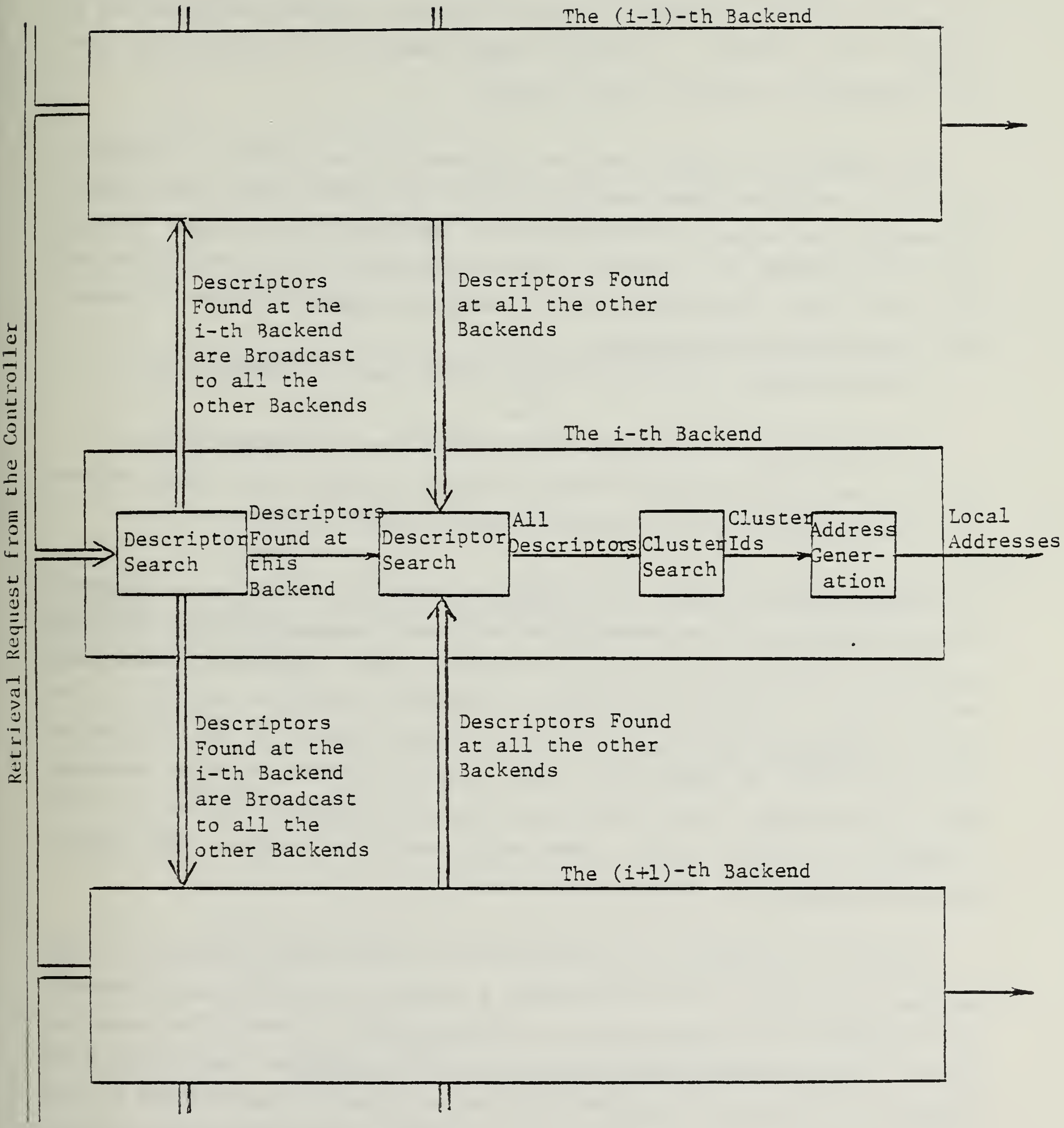

The (i-1)-th Backend
Descriptors Found at the i-th Backend are Broadcast to all the other Backends
Descriptors Found at all the other Backends
The i-th Backend
Retrieval Request from the Controller
Descriptor Search
Descriptors Found at this Backend
Descriptor Search
All Descriptors
Cluster Search
Cluster Ids
Address Gener-ation
Local Addresses
Descriptors Found at the i-th Backend are Broadcast to all the other Backends
Descriptors Found at all the other Backends
The (i+1)-th Backend

Figure 4. Overview of Directory Management as Seen From The i-th Backend

casts the resulting descriptor ids to the other backends. After the descriptor ids from all other backends have been received, cluster search is used to determine the clusters. Finally, address generation determines the local disk addresses for records at that backend.

The backend can do more than just retrieve all the records in a cluster. First, it can select those and only those records that satisfy the request. For example, the request to find records of employees in Columbus earning more than $20,000 but not more than $28,000 and with more than 10 but not more than 15 years experience, requires _selecting records_ from two clusters. Those clusters are identified by

(CITY=Columbus) and (20001=<SALARY=<25000) and (11=<SERVICE=<15)
and
(CITY=Columbus) and (25001=<SALARY=<30000) and (11=<SERVICE=<15).

All the records will be selected from the first cluster, but only records with SALARY=<$28,000 will be selected from the second cluster.

Often, not all the data in a record is needed to respond to a request. In this example, only the names of the employees might be required. Thus the appropriate values must be extracted from the record. The other values may be discarded. Although not shown in this example, MDBS can perform various types of _aggregate operations_ on a set of values instead of just returning the raw values. An example would be to find the average salary of employees who live in Columbus. Thus after selecting the appropriate records and extracting the salary values, MDBS would compute the average. The steps of record processing are summarized in Figure 5.

Referring to Figure 6, the execution of a user request can now be summarized as follows. The user submits a request to the host which transmits that request, in an internal form, to the controller of MDBS. The controller parses the request and then broadcasts it to the backends. Each backend determines its portion of the descriptor ids and broadcasts the results to the other backends. Each backend also determines the clusters that must be searched and the corresponding local disk addresses. Then the appropriate records are selected, values extracted and results sent back to the controller. When the controller has received the results from all the backends, it

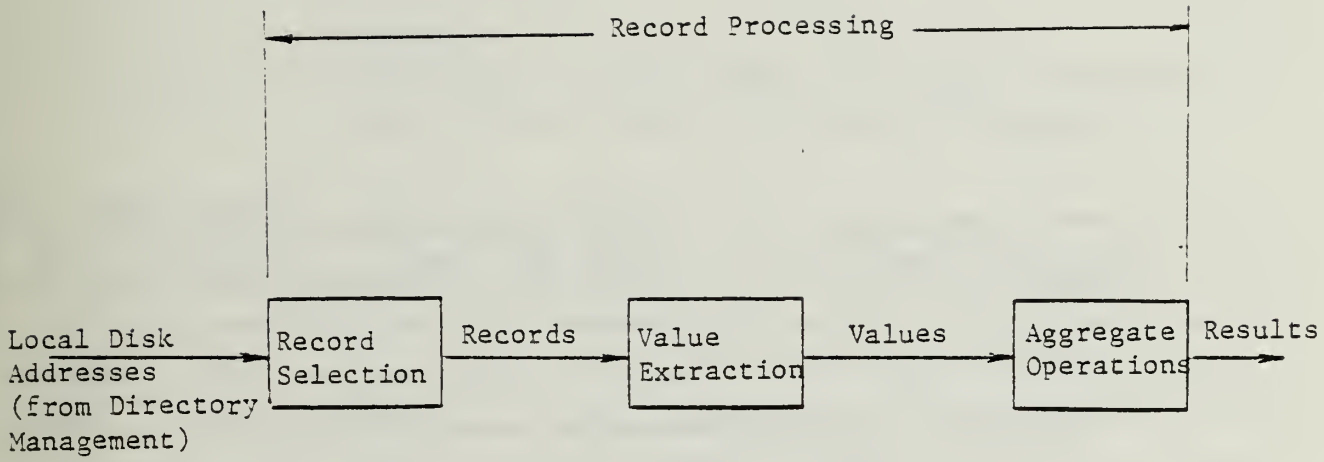
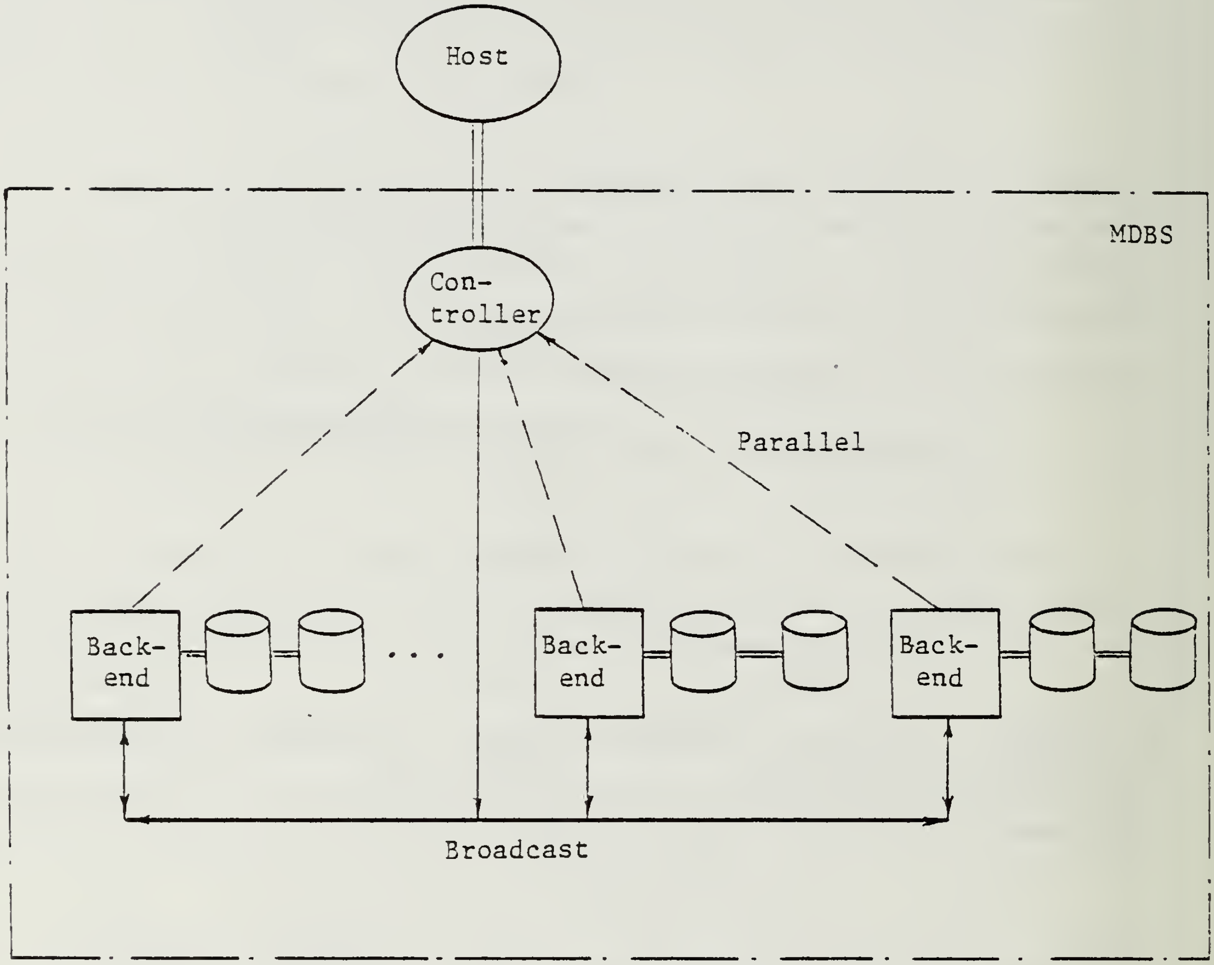


Figure 5. Record Processing Function



Broadcast Mode

- Controller-to-all-backends operation (e.g., query)
- Backend-to-all-other-backends operations (e.g., transferring descriptor ids)

Parallel Mode

- Response-of-each-backend-to-controller operations (e.g., forwarding retrieved data)

Figure 6. Modes of MDBS Operations

performs any aggregate operation required and then forwards the final results to the host for return to the user.

1.2 The Implementation Strategy - What and Why?

It seems only reasonable to develop most systems in stages. For prototype systems such an approach seems even more important. Thus we have planned to develop several versions of MDBS. We chose to begin with an implemention of a very simple system.

1.2.1 Version I - A Very Simple System: Single Mini Without Concurrency Control and With Simplified Directory Management

We started the implementation effort with a system which was intended to be as simple as possible. The aim was to get something running so that we could gain some experience with both the MDBS design and our new computer systems. Thus we had chosen to simplify the design as much as possible. MDBS-I, which is in the final stages of implementation, executes only a single request at a time. It runs on a single computer. There is no distinction made about the slave and master. In other words, there is no separate controller. Directory management is simplified by storing all directory data in the main memory. There are no concurrent execution of requests. Since the whole system runs as a single operating system process, the interface with the operating system is minimized.

1.2.2 Version II - A Simple System: Single Mini With Concurrency Control

The second version, whose details have been designed and the implementation effort is under way, will allow concurrent execution of requests, but will still be restricted to a single mini. We plan to use the services of our operating system to facilitate this concurrent processing. Thus we will use the capability of creating independent concurrent processes which communicate among themselves. These processes will execute in parallel so that MDBS-II will be able to execute requests in parallel. This version will allow us to gain experience with the problem of multiple processes and the problem of concurrency control.

1.2.3 Version III - The First "Real" System : Multiple Minis With Concurrency Control

After MDBS-II is working, we will transfer the system to our real environment including a controller (i.e., VAX 11/780) and several backends (PDP 11/44s). This transfer should be fairly easy, since the major changes required will be to replace communications between processes in one computer by communications between processes running on different computers. This version will allow us to see how the intercomputer communication overhead is going to affect system performance. This system, MDBS-III, will still not be sufficient for a full MDBS, since it has a very simplified directory management subsystem. However, it will allow us to begin preliminary testing of the MDBS design.

1.2.4 Version IV - The Real System With "Good" Directory Management

This version, whose design details are being proposed, will include a fully implemented directory management subsystem utilizing the secondary memories. it will be a complete prototype system, except for the lack of access control features. This system, MDBS-IV, will be the one on which we will try to validate the simulation studies used in the development of the original design.

1.2.5 Version V - The Full System With All the Designed Features Included

The final version will incorporate access control in the backends and a friendly user-interface in the controller or host computer.

1.3 An Overview of the MDBS Implementation

In this section, we give an overview of the implementation effort to date. The recent implementations are described in more detail in later chapters. Details on our earlier implementation effort can be found in [Kerr82].

1.3.1 A Top-level View of MDBS

The MDBS is viewed in terms of controller functions and backends functions (see Figure 7). In the following two sections, we describe the functions performed in the controller and the backends, respectively. Then we will describe the process of request execution for four types of request: delete, insert, retrieve and update.

There are, however, some essential functions which are not included in either of these divisions. Among these are system generation, system startup/shutdown, and other system utilities such as database load, file generation and database reorganization. These functions will generally be initiated in the minicomputer which serves as the MDBS controller. They are not, however, a logical part of the major functions of the controller.

1.3.2 Functions of the Controller

The MDBS controller consists of three categories of functions: request preparation, insert information generation and post processing (see Figure 7 again). The <u>request preparation functions</u> are those which must be performed before a request or a transaction can be broadcasted to the backends. For example, each request must be parsed and checked for syntax errors before it can be broadcasted to the backends. The <u>insert information generation functions</u> are those which must be performed during the processing of an insert request to furnish additional information required by the backends. For example, a backend should be selected for storing the record being inserted into the secondary storage of the backend. The <u>post processing functions</u> are those which must be performed after replies are returned from the backends, but before the results of a request or a transaction are forwarded to the host machine. For example, the results for a request returned by each backend should be collected. After receiving the results from each backend, the response to the request can be sent to the host machine.

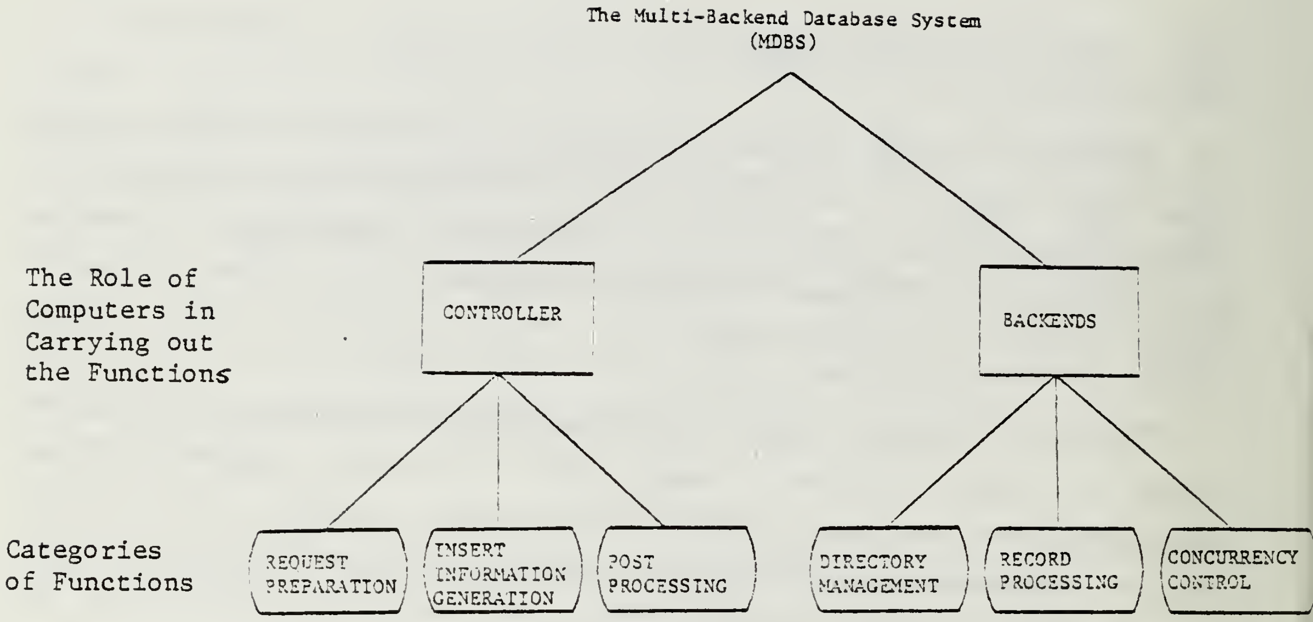


Figure 7. The MDBS Structure

We note that there are no concurrency control functions in the controller. Since user requests are carried out by the backends, there is no need for concurrency control in the controller. The controller must only associate sequence numbers with the user requests.

1.3.3 Functions of each Backend

Each backend in MDBS consists of three categories of functions: directory management, record processing and concurrency control (also in Figure 7). The directory management functions perform descriptor search, cluster search, address generation and directory table maintenance. For example, these functions find the ids of descriptors corresponding to a set of predicates (keywords), determine the cluster id corresponding to a set of descriptors and determine the secondary storage addresses of the records in a cluster. The record processing functions perform record storage, record retrieval, record selection and attribute value extraction of the retrieved records. For example, these functions store records into the secondary storage, retrieve records from the secondary storage and select the records that satisfy a query from a set of records. The concurrency control functions perform operations which ensure that the concurrent and interleaved execution of user requests will keep the database consistent. For example, these fuctions schedule a user request for execution based on the set of clusters needed by the request. In this chapter, we do not consider concurrent and interleaved execution of user requests. The concurrency control mechanism is described in Chapter 4.

1.3.4 Request Execution in MDBS

In this section, we describe briefly the sequence of actions taken by MDBS in executing insert requests and non-insert requests (delete, retrieve and update). The sequence of actions is described in terms of flow of data and in terms of the functions categorized above. The sequenece of actions taken by MDBS in executing each of the four types of request: insert, delete, retrieve and update is described in more detail in the later chapters.

(A) Sequence of Actions for an Insert Request

The sequence of actions for an insert request is shown in Figure 8. Some flow of data is common to all types of request, shown as dotted lines in the figure. Thus, we first describe these common data flows. The arrow entering Request Preparation indicates that a request or a transaction is the input to this module. The input comes from the host machine. Request Preparation sends the number of requests in a transaction to Post Processing. The number of requests in a transaction is used by Post Processing to determine whether processing of the transaction is complete. Request Preparation also sends a request (transaction) along with error messages to Post Processing if the request (transaction) has syntax errors. Post Processing collects all the results related to a request (transaction) and sends the results to the host machine. The arrow leaving Directory Management indicates that the descriptor ids found by a backend are sent to the other backends. The arrow entering Directory Management indicates that the descriptor ids found by the other backends are sent to this backend.

We now describe the flow of data specific to insert requests, shown as solid lines in Figure 8. After receiving, parsing and formatting a request, Request Preparation sends the formatted request to Directory Management in the backends. We recall that the record part of the request consists of many keywords and each backend performs the descriptor search for a different set of keywords in the record. Thus, Directory Management at a backend finds the ids of descriptors corresponding to the set of keywords to be processed at the backend and broadcasts the ids to the other backends. After receiving the descriptor ids sent by the other backends, Directory Management determines the cluster id, if any, of the cluster to which the record belongs. It then sends the cluster id to Insert Information Generation in the controller. Insert Information Generation determines the backend at which the record is to be inserted and broadcasts a message to Directory Management in the backends. The backends that are not to insert the record discard the record. Directory Management in the backend that is to insert the record determines the secondary storage address for inserting the record. That address and the formatted request are then passed to Record Processing. Record Processing stores the record into the secondary storage and sends a completion signal to

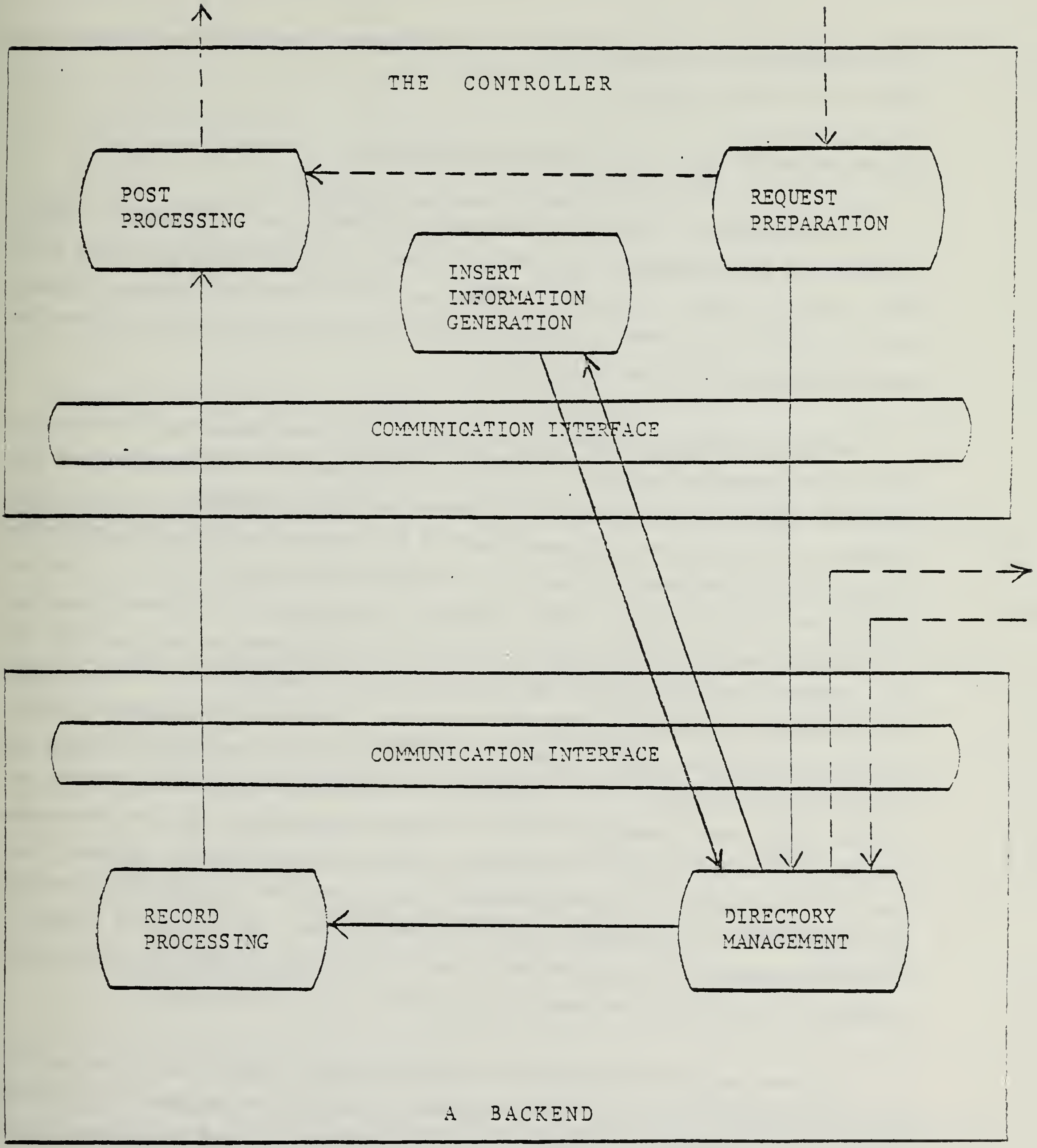


Figure 8. Sequence of Actions for an Insert Request

Post Processing in the controller. Post Processing then sends a completion signal to the host machine.

(B) Sequences of Actions for Non-insert Requests

The sequences of actions for non-insert requests are all similar. Thus, we describe the sequence of actions only for a retrieve request in this section. This is shown in Figure 9. (Here, we assume that the retrieve request was not caused by an update request. Details on retrieve requests caused by update requests are given in Chapter 2.)

Request Preparation, after receiving, parsing and formatting a request, sends the formatted request to Directory Management in the backends and the aggregate operators, if any, in the request to Post Processing in the controller. We recall that the query part of the request consists of many predicates and each backend performs the descriptor search for a different set of predicates in the query. Thus, Directory Management at a backend finds the ids of descriptors corresponding to the set of predicates to be processed at the backend and broadcasts the ids to the other backends. After receiving the descriptor ids sent by the other backends, Directory Management determines the cluster ids. Finally, it determines the secondary storage addresses of the records in the clusters so identified and sends the record addresses and the formatted request to Record Processing. Record Processing fetches the records from the secondary storage and selects the records that satisfy the query. It then extracts the values from the selected records. If aggregation is not needed, Record Processing sends the extracted values to Post Processing in the controller. Post Processing collects all the results related to the request and sends the results to the host machine.

If some aggregations are to be applied, Record Processing, after selecting the records and extracting the values, applies the aggregate operations on the set of values. It then sends the results to Post Processing in the controller. The partial results from all the backends are collected in Post Processing. Post Processing performs the aggregate operations on the partial results and sends the results to the host machine.

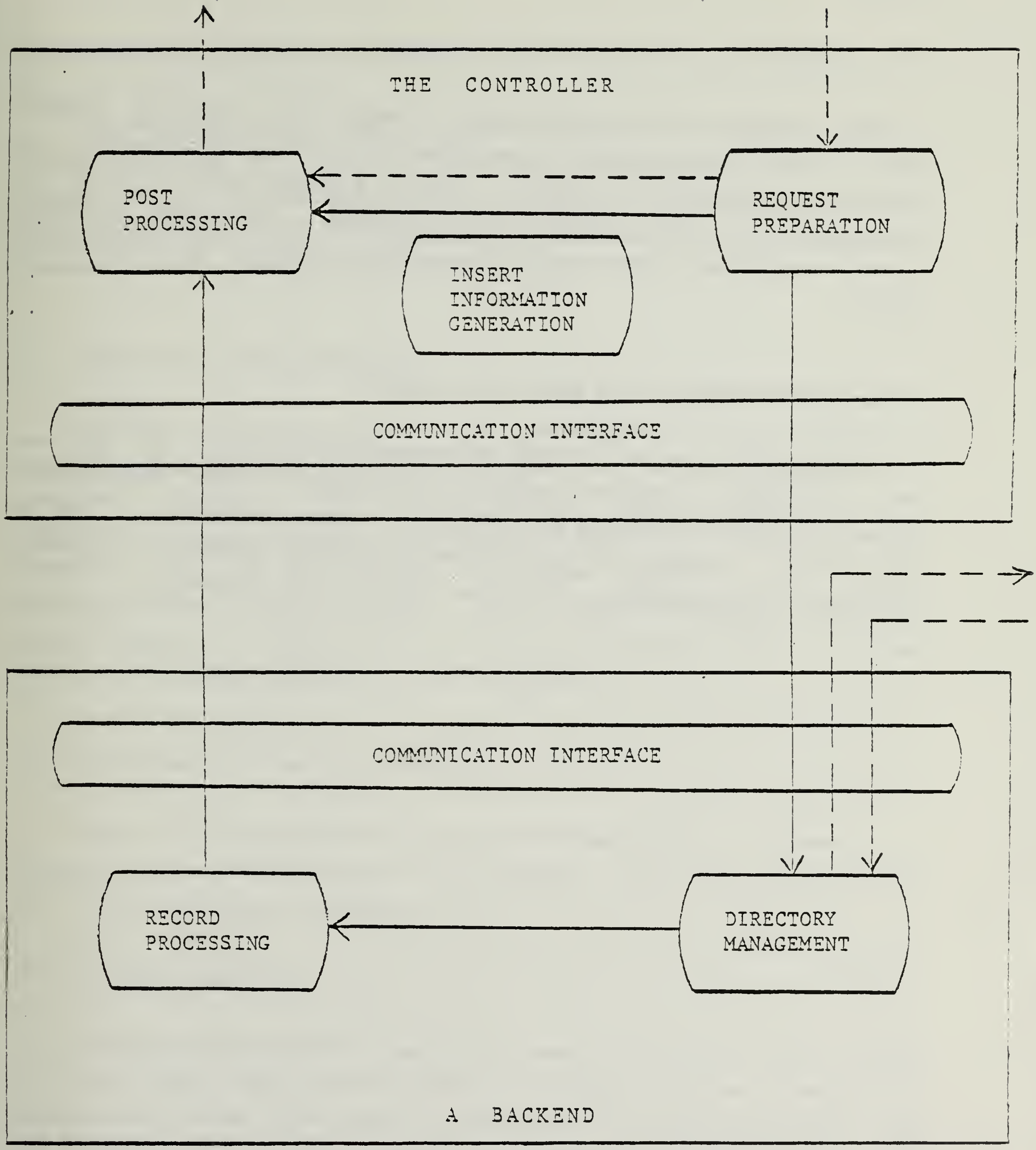


Figure 9. Sequence of Actions for a Retrieve Request

1.3.5 The Role of the Communication Interface

Let us now describe the boxes labeled Communication Interface in Figures 8 and 9. They provide the mechanism for communications between two functions in two different computers. There is a communication interface in each computer, i.e., the controller and the backends, since certain functions in each computer must communicate with certain functions in the other computers.

1.4 The Organization of the Rest of the Report

We describe in detail the MDBS implementation in the rest of this report. In Chapter 2, we give a functional description of MDBS. The controller and the backends functions are described in detail in Chapters 3 and 4, respectively. A method for testing MDBS is described in Chapter 5. Finally in Chapter 6, we summerize our software engineering experience.

2.0 A FUNCTIONAL DESCRIPTION OF MDBS

As described in the previous chapter and depicted in Figure 7, MDBS is viewed in terms of controller functions and backend functions. In this chapter, we describe the functions of the controller and backends in detail. We also describe in detail the process of request execution for four types of request: delete, insert, retrieve and update.

2.1 Functions of the Controller

The MDBS controller functions are considered in three categories: request preparation, insert information generation and post processing. (As described in Chapter 1, there are no concurrency control functions in the controller.) In the following, we describe the functions of each of the three categories. We will not discuss the system functions such as system startup/shutdown in this section. We will describe a package for testing MDBS in Chapter 5. Other details on the system functions such as database load and file generation can be found in [Kerr82].

2.1.1 The Request Preparation Functions

These are the functions which must be perfomed before a request or a transaction can be broadcasted to the backends. The names of the functions are: Parser and Request Composer.

(A) The Parser Function

This function parses the requests and checks for syntax errors. Input to Parser comes from the host machine. The input is either a request or a transaction. If the input request (transaction) is parsed correctly, then the parsed request (parsed transaction) is passed to Request Composer. If the input request (transaction) contains syntax errors, Parser returns the request (transaction) along with error messages to Reply Monitor. MDBS does not execute a transaction unless all the requests in the transaction are parsed correctly, i.e., a transaction is rejected if one or more requests contain syntax errors.

For retrieve requests with aggregate operators, Parser sends the type of aggregate operators (AVG, MAX, MIN, SUM, COUNT) to Aggregate Post Operation where the specific aggregate operations are to be performed on the partial results to be returned by the backends.

When the input to Parser is a transaction, Parser passes the number of requests in the transaction to Reply Monitor. The number of requests in a transaction is used by Reply Monitor to determine whether the processing of the transaction is complete.

(B) The Request Composer Function

Before describing this function, let us review the update requests in MDBS. The syntax of an update request is:

UPDATE Query Modifier

where the modifier specifies the kinds of modification that need to be done on records that satisfy the query. The modifier may be one of the following five types:

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

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

An example of a type-0 modifier is:

<SALARY=50000>

This sets the salary in all the records being modified to 50000.

An example of a type-I modifier is:

<SALARY=1.1*SALARY>

This raises the salary in all the records being modified by 10%.

An example of a type-II modifier is:

<MONTHSAL=YEARSAL/12>

This sets the monthly salary in all the records being modified to be a twelfth of their own yearly salaries.

An example of a type-III modifier is:

<SALARY=SALARY of (FILE=Wife) and (NAME=Tara)>.

This causes the following actions to be taken by MDBS. Using the query "(FILE=Wife) and (NAME=Tara)", a record is retrieved. Then, the SALARY value of that record is obtained. This value is used for the salary in all the records being modified.

An example of a type-IV modifier is:

<SALARY=SALARY of 2000>

which modifies the salary in all the records being modified to that of the record stored in location 2000. In order to use this type of modifier, the user must have previously issued a retrieve request which had WITH POINTER option. We note that, in order to execute an update request containing a type-III or type-IV modifier, a record must first be retrieved by MDBS on the basis of a user-provided query or pointer. We now describe the Request Composer function.

This function transforms a parsed request into the form required for processing at the backends. Request Composer receives each parsed request (parsed transaction) from Parser. For all requests except updates with type-III or type-IV modifier, Request Composer formats the request and sends it to the backends for processing. For update requests with type-III or type-IV modifier, Request Composer first generates a retrieve request. It then saves all the information necessary to generate an update request with type-0 modifier when the value from the retrieve request is received. When the value is received from a backend, the update request with type-0 modifier will be generated and sent to the backends.

Processing an update request may cause one or more updated records to change cluster. When this occurs, the old records should be marked for deletion and the updated records should be inserted. Request Composer initiates the actions required for the insertion of the updated records that change cluster.

2.1.2 The Insert Information Generation Functions

These are the functions which must be performed during the processing of an insert request to furnish additional information required by the backends. The names of the functions are: Backend Selector, Cluster Id Generator and Descriptor Id Generator.

(A) The Backend Selector Function

When processing an insert request, Backend Selector determines the backend at which the record is to be inserted. The backend selection is based on the criterion that the records in each cluster should be distributed among the backends. (As described in Chapter 1, the records in each cluster are spread across the backends to allow the records in the cluster to be accessed in parallel.)

(B) The Cluster Id Generator Function

In order to save storage and time, each cluster is identified by a cluster id, instead of being identified by a set of descriptors which characterize the cluster. Cluster Id Generator produces a new cluster id for a new cluster.

(C) The Descriptor Id Generator Function

To further save storage and time, each descriptor is also identified by a descriptor id, instead of being identified by an attribute and its attribute value (attribute value ranges). Descriptor Id Generator produces a new descriptor id for a new descriptor.

2.1.3 The Post Processing Functions

Before the results of a request or a transaction are forwarded to the host machine, these functions must be performed on the replies returned by the backends. The names of the functions are: Aggregate Post Operation and Reply Monitor.

(A) The Aggregate Post Operation Function

When there is an aggregate operator in a retrieve request, each backend performs the aggregate operation on those records in that backend satisfying the query. The partial aggregate results are sent to Aggregate Post Operation by the backends. Parser sends the type of aggregate operator (AVG, MAX, MIN, SUM, COUNT) to Aggregate Post Operation where the partial results are received from the backends and are combined to give the final result of the specific aggregate operation. The results are then forwarded to Reply Monitor.

(B) The Reply Monitor Function

This function collects all the results for a request or a transaction, and forwards them to the host machine. As described earlier, Parser sends the number of requests in a transaction to Reply Monitor. Reply Monitor uses this number to determine whether the processing of the transaction is complete.

2.2 Functions of each Backend

Each backend in MDBS consists of three categories of functions: directory management, record processing and concurrency control (see Figure 7 again). (As in Chapter 1, we do not consider concurrent and interleaved execution of user requests in this chapter. We describe the concurrency control mechanism in Chapter 4.) In the following sections, we describe the functions of each of the first two categories, i.e., directory management and record processing.

2.2.1 The Directory Management Functions

These functions perform directory operations such as cluster determination, address generation and directory table maintenance. The names of the functions are: Descriptor Search, Cluster Search and Address Generation.

(A) The Descriptor Search Function

This function determines the descriptor ids of the descriptors that satisfy the predicates (keywords) in a query (record). Input to Descriptor Search comes from Request Composer in the controller, in the form of a formatted request. As described in detail in [Hsia81a], if there are N backends processing a query (record) with X predicates (keywords), then each backend performs the descriptor search on X/N predicates (keywords) and broadcasts the descriptor ids to the other backends.

(B) The Cluster Search Function

This function determines either the cluster id of the cluster to which a record belongs (for an insert request) or the cluster ids of the clusters whose records satisfy a query (for a non-insert request). Input to Cluster Search are the descriptor ids found by Descriptor Search in all the backends. For insert requests, Cluster Search passes the cluster id, if any, to Backend Selector in the controller. For non-insert requests, the cluster ids are passed to Address Generation.

(C) The Address Generation Function

This function determines either the secondary storage address for storing a record when processing an insert request or the addresses of all the records in a set of clusters when processing a non-insert request. For insert requests, Backend Selector in the controller determines which backend is to insert the record. When a backend is selected, Address Generation in that backend determines the secondary storage address for record insertion. That address and the formatted request are then passed to Physical Data Operation.

For non-insert requests, Cluster Search passes the cluster ids to Address Generation. Address Generation finds the addresses of the records in these clusters and passes the addreses and the formatted request to Physical Data Operation.

2.2.2 The Record Processing Functions

These functions perform operations such as record selection and field extraction of the retrieved records. The names of the functions are: Physical Data Operation and Aggregate Operation.

(A) The Physical Data Operation Function

Input to this function comes from Address Generation. The input is a set of secondary storage addresses and a formatted request. Physical Data Operation performs different actions depending on the type of the request. For an insert request, Physical Data Operation stores the record being inserted into the secondary storage.

For a non-insert, i.e., delete, retrieve or update, Physical Data Operation fetches the records from the secondary storage and selects the records that satisfy the query in the request. It then performs the intended operation on the basis of the type of the non-insert request. For delete requests, Physical Data Operation marks the selected records for deletion.

For retrieve requests, Physical Data Operation extracts the values from the selected records and passes the values either to Aggregate Operation, if an aggregation is to be applied, or to Reply Monitor, if aggregation is not needed. For retrieve requests caused by update requests with type-III or type-IV modifier, Physical Data Operation sends the results to Request Composer in the controller. The results will be used in the controller to form update requests with type-0 modifier from the update requests with type-III or type-IV modifier.

For update requests, Physical Data Operation updates the selected records and returns to the secondary storage those updated records that have not changed cluster. If one or more records change cluster, Physical Data Operation marks the old records for deletion and sends the records that have changed cluster to Request Composer in the controller. Request Composer initiates the actions required for the insertion of these records into their new clusters.

(B) The Aggregate Operation Function

This function performs the partial aggregate operations in retrieve requests. Input to Aggregate Operation comes from Physical Data Operation in the form of a set of values and the aggregate operators to be applied. Aggregate Operation applies the aggregate operations on the set of values and passes the results to Aggregate Post Operation in the controller.

2.3 <u>Request Execution in MDBS</u>

In this section, we describe in detail the sequence of actions taken by MDBS in executing each of the four types of request: insert, delete, retrieve and update. As in Chapter 1, the sequence of actions is described in terms of flow of data and in terms of functions presented earlier.

2.3.1 Sequence of Actions for Insert Requests

The sequence of actions for an insert request is shown in Figure 10. As in Chapter 1, we first describe the flow of data common to all types of request, shown as dotted lines in Figure 10. The arrow entering Parser indicates that a request or a transaction is the input to this function. The

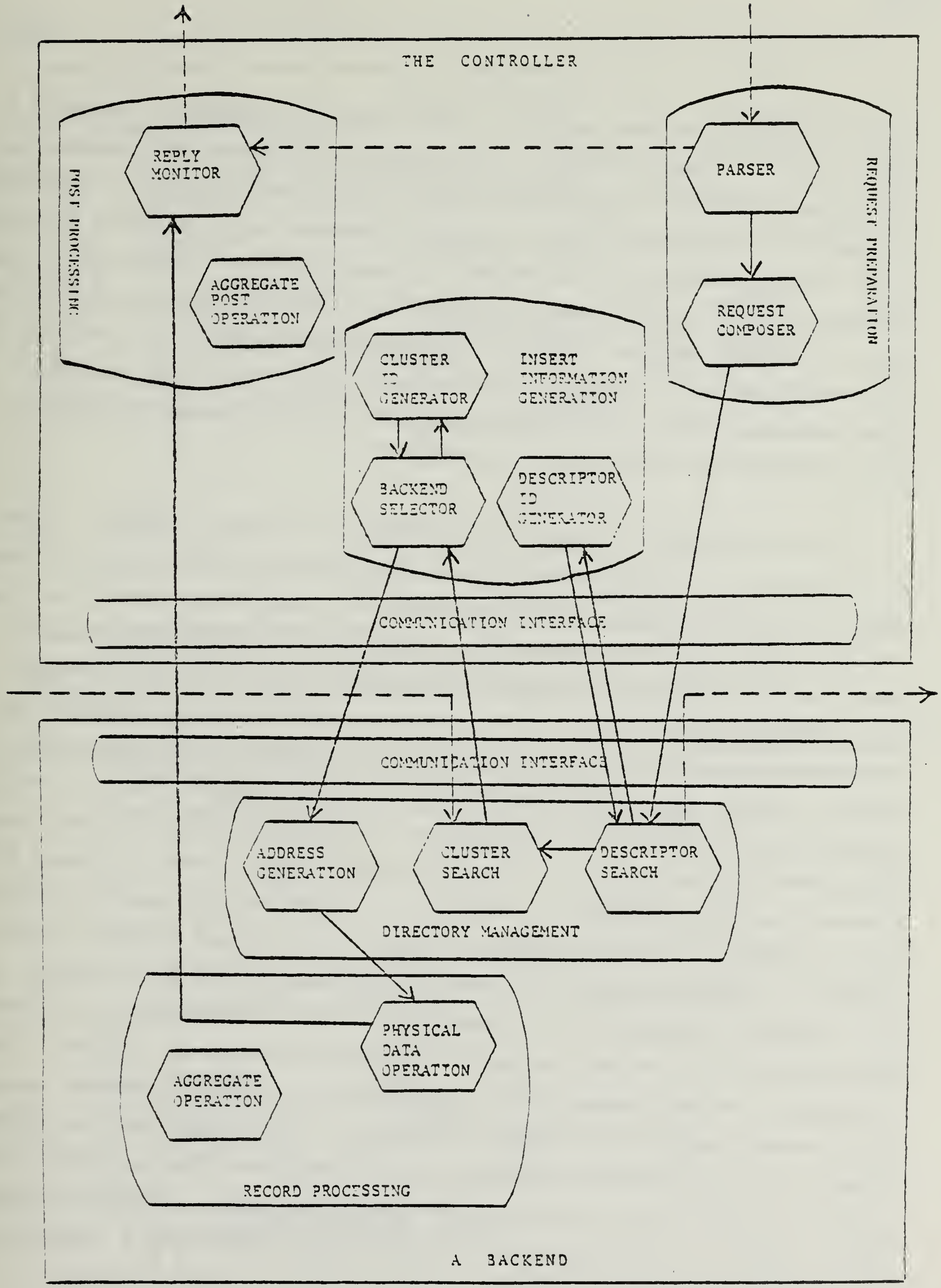


Figure 10. Sequence of Actions for an Insert Request

input comes from the host machine. Parser sends the number of requests in a transaction to Reply Monitor. The number of requests in a transaction is used by Reply Monitor to determine whether the processing of the transaction is complete. Parser also sends a request (transaction) along with error messages to Reply Monitor if the request (transaction) has syntax errors. Reply Monitor collects all the results related to a request (transaction) and sends them to the host machine. The arrow leaving Descriptor Search indicates that the descriptor ids found by a backend are sent to the other backends. The arrow entering Cluster Search indicates that the descriptor ids found by the other backends are sent to this backend.

We now describe the flow of data specific to insert requests, shown as solid lines in Figure 10. Parser, after receiving and parsing a request, sends the parsed request to Request Composer. After transforming the parsed request into the form required for processing at the backends, Request Composer sends the formatted request to Descriptor Search in the backends. We recall that the record part of the request consists of many keywords and each backend performs the descriptor search for a different set of keywords in the record. Thus, Descriptor Search at a backend finds the ids of descriptors corresponding to the set of keywords to be processed at the backend, broadcasts the ids to the other backends and forwards them to Cluster Search. Cluster Search determines the cluster id, if any, of the cluster to which the record belongs. It then sends the cluster id to Backend Selector in the controller. Backend Selector determines the backend at which the record is to be inserted and broadcasts a message to Address Generation in the backends. The backends that are not to insert the record discard the record. Address Generation in the backend that is to insert the record determines the secondary storage address for storing the record. That address and the formatted request are then passed to Physical Data Operation. Physical Data Operation stores the record into the secondary storage and sends a completion signal to Reply Monitor in the controller. Reply Monitor then sends a completion signal to the host machine.

2.3.2 Sequence of Actions for Delete Requests

The sequence of actions for a delete request is shown in Figure 11.

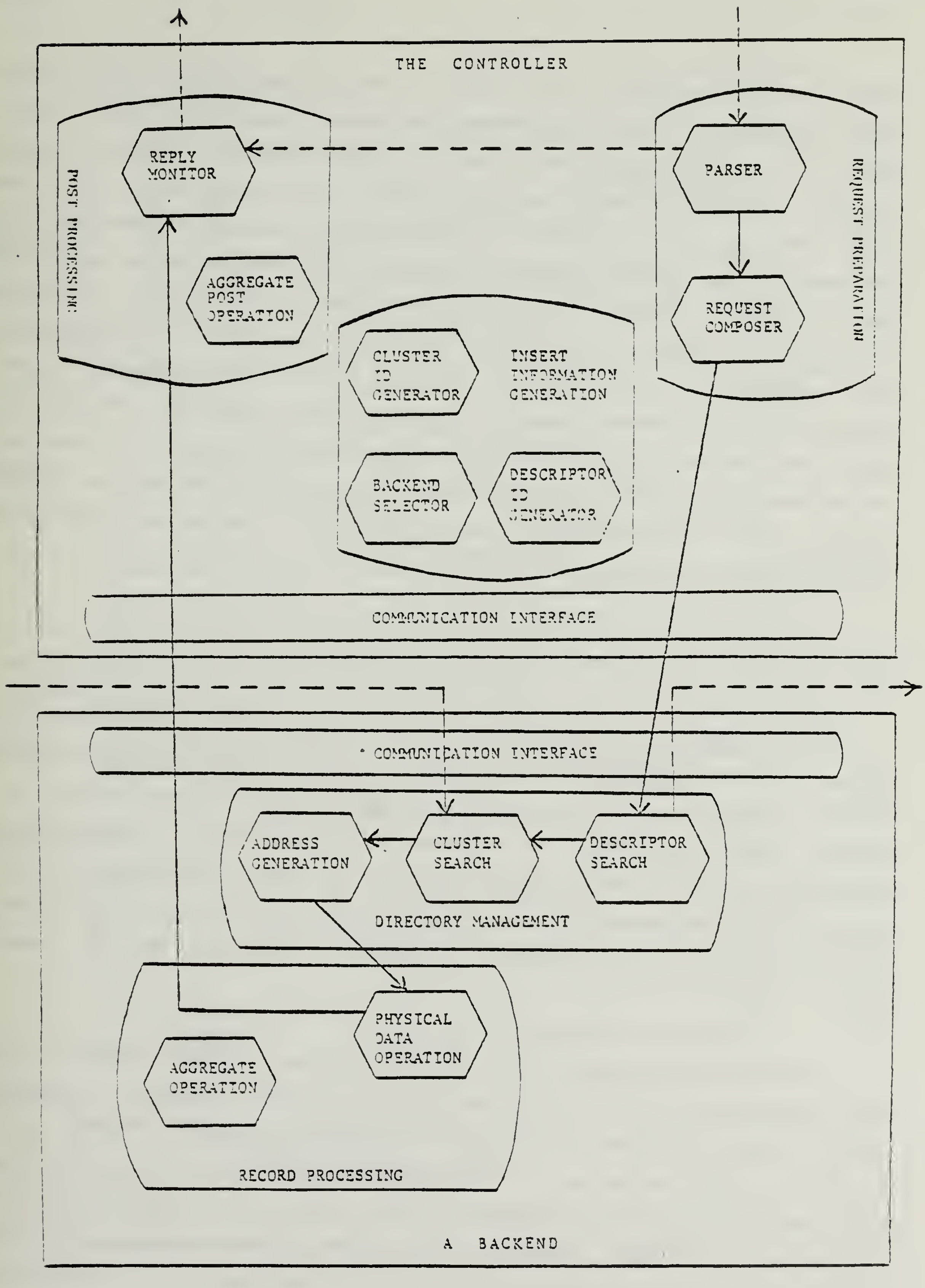


Figure 11. Sequence of Actions for a Delete Request

Parser, after receiving and parsing a request, sends the parsed request to Request Composer. After transforming the parsed request into the form required for processing at the backends, Request Composer sends the formatted request to Descriptor Search in the backends. Descriptor Search at a backend finds the ids of descriptors corresponding to the set of predicates to be processed at the backend, broadcasts them to the other backends and forwards them to Cluster Search. Cluster Search determines the cluster ids and gives them to Address Generation. Address Generation determines the secondary storage addresses of the records in these clusters and sends the record addresses and the formatted request to Physical Data Operation. Physical Data Operation fetches the records from the secondary storage. It then selects the records that satisfy the query, marks the selected records for deletion, returns them to the secondary storage and sends a completion signal to Reply Monitor in the controller.

2.3.3 Sequence of Actions for Retrieve Requests

The sequence of actions for a retrieve request is shown in Figure 12. Parser, after receiving and parsing a request, sends the parsed request to Request Composer and the aggregate operators, if any, in the request to Aggregate Post Operation. The sequence of actions taken by Request Composer, Descriptor Search, Cluster Search, Address Generation and Physical Data Operation (up to the selection of the records that satisfy the query) is the same as the other non-insert request, i.e., delete. Thus, we do not repeat it here.

If the retrieve request was not caused by an update request, Physical Data Operation extracts the values from the selected records. If aggregation is not needed, Physical Data Operation sends the extracted values to Reply Monitor in the controller. If some aggregations are to be applied, Physical Data Operation passes the extracted values along with the aggregate operators to Aggregate Operation. Aggregate Operation applies the aggregate operations on the set of values and sends the results to Aggregate Post Operation in the controller. The partial aggregate results from all the backends are collected in Aggregate Post Operation. Aggregate Post Operation performs the aggregate operations on the partial results. The results are then forwarded to

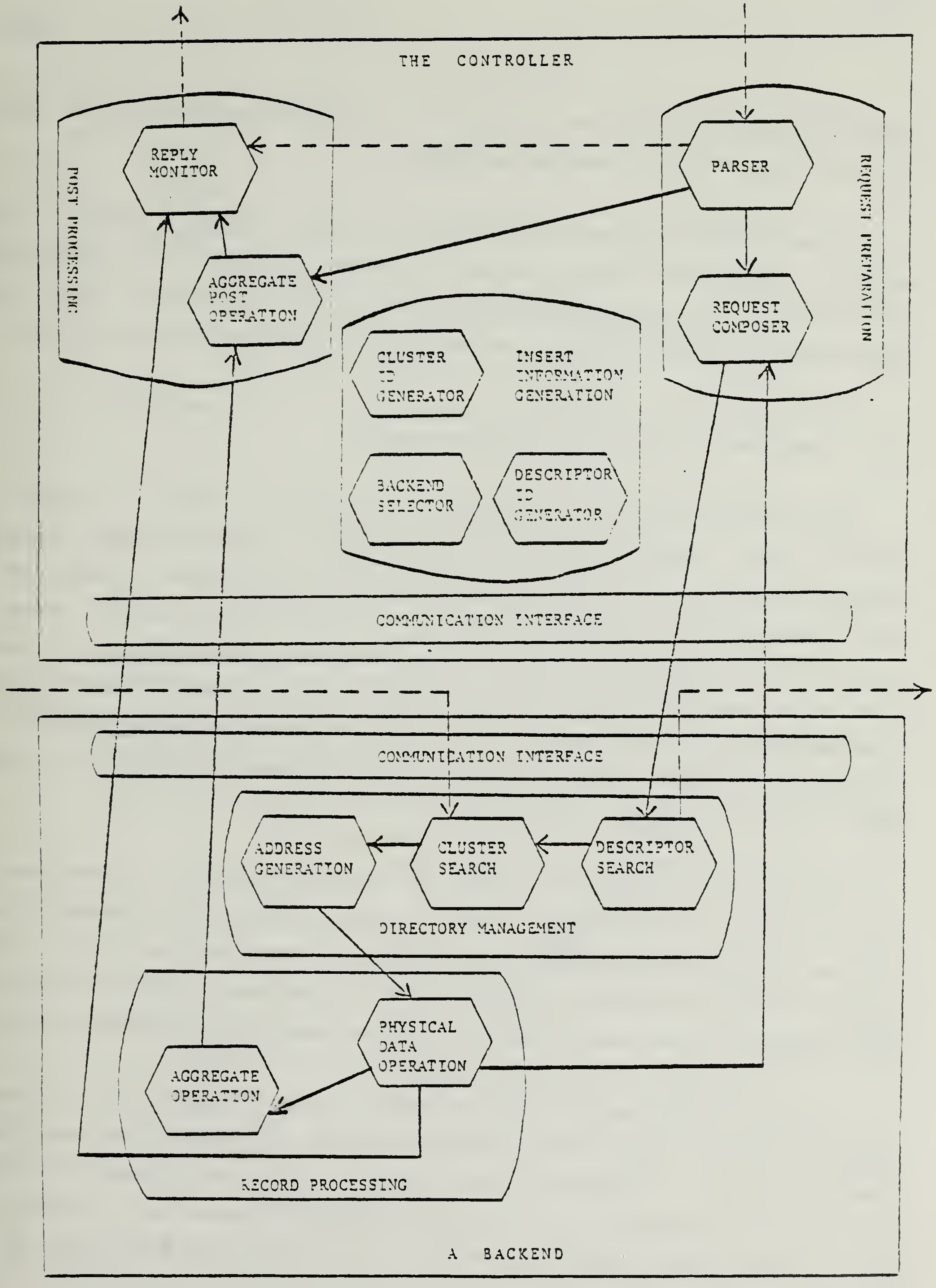


Figure 12. Sequence of Actions for a Retrieve Request

Reply Monitor. Reply Monitor collects all the results related to the request and sends the results to the host machine.

If the retrieve request was caused by an update request, Physical Data Operation sends the result, if any, to Request Composer in the controller. (The results will be used in the controller to form an update request with type-0 modifier from the update request with type-III or type-IV modifier.)

2.3.4 Sequence of Actions for Update Requests

The sequence of actions for an update request is shown in Figure 13. Parser, after receiving and parsing a request, sends the parsed request to Request Composer. The sequence of actions will be different depending on the type of modifier in the update request. We first describe the case where the modifier is not type-III or type-IV. In this case, the sequence of actions taken by Request Composer, Descriptor Search, Cluster Search, Address Generation and Physical Data Operation (up to the selection of the records that satisfy the query) is the same as the other non-insert request, i.e., delete. Thus, we do not repeat it here.

Physical Data Operation updates the selected records. It then uses Descriptor Search to determine which updated records have changed cluster. Physical Data Operation stores those updated records that have not changed cluster into the secondary storage. It will then send a completion signal to Reply Monitor in the controller if no updated record has changed cluster.

If one or more updated records change cluster, Physical Data Operation marks the old records for deletion and sends the records that have changed cluster to Request Composer in the controller. Request Composer initiates the actions required for the insertion of these records into their new clusters. After these records are inserted, the original update request is complete.

If the modifier in the update request is type-III or type-IV, Request Composer in the controller first generates a retrieve request. It then saves all the information necessary to generate an update request with type-0 mod-

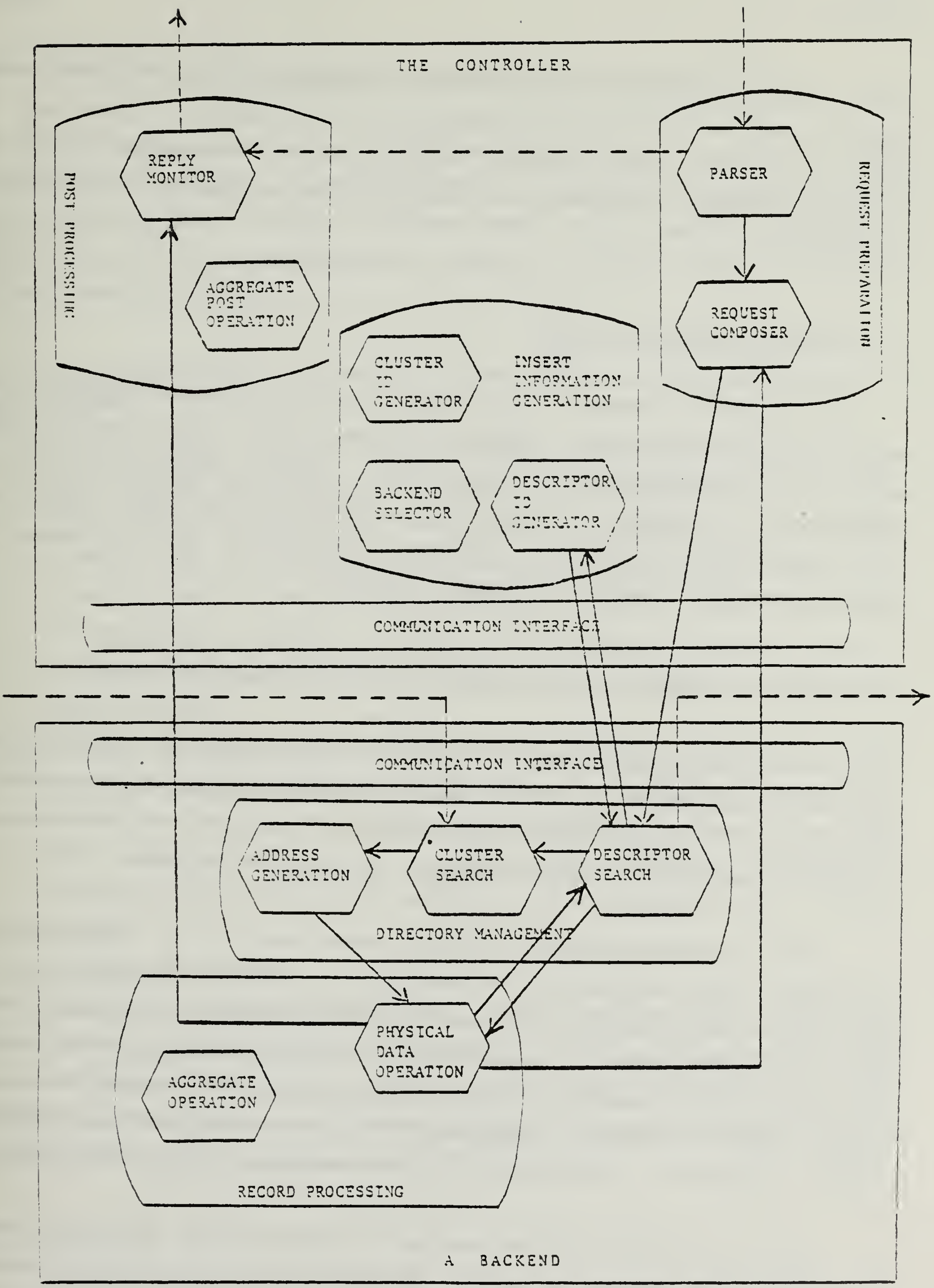


Figure 13, Sequence of Actions for an Update Request ifier when the value from the retrieve request is received. When Request Composer receives the value from Physical Data Operation, it generates the update request with type-0 modifier and sends it to the backends. After this new update request is executed to completion, the original update request is complete.

2.4 Process Structure of MDBS

Most operating systems provide mechanisms for allowing concurrent execution of different processes. These mechanisms include primitives for communication and synchronization among processes. Process communication and synchronization primitives of the operating system are the basic system primitives that MDBS-II utilizes for concurrent execution of multiple requests as well as concurrent control of common resources.

2.4.1 Two Alternative Process Structures for Implementing MDBS

Process and synchronization primitives provided by the operating systems may be characterized as either message-oriented or procedure-oriented, depending on how they implement the notion of process and synchronization [Laue79]. We could use either approach for implementing MDBS.

Using a message-oriented approach, there would be a fixed number of processes (one process per MDBS activity). Directory management, for example, may be implemented as a process. Synchronization of directory management activities may be implemented by passing messages among processes. There would be a relatively limited amount of direct sharing of data in the memory among processes. Processes for each activity would be created when MDBS is started up. They would be deleted only when MDBS is shut down.

Using a procedure-oriented approach, there would be a varying number of processes (one process per user). Synchronization of user activities may be implemented by direct sharing and locking of common data in the main memory. Processes would be rapidly created and deleted.

2.4.2 The Choice of Message-oriented Approach to Implement MDBS

The functional composition of MDBS described in the previous sections allows either approach, message-oriented or procedure-oriented, to be used for implementing MDBS. However, we have chosen to use message-oriented approach for the first implementation of MDBS-II. In this section, we give the rationale behind our choice.

There are two major problems associated with 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 resource, all other processes will be queued up behind the locked resource. Thus, the database system performance will be degraded.

The real-time operating system, RSX11, being used in MDBS facilitates message passing. It also allows a process to receive messages from multiple proceses. Because of the aforementioned two problems with the procedure-oriented approach and because of the environment provided by RSX11, we have decided to use the message-oriented approach.

3.0 AN IMPLEMENTATION OF THE CONTROLLER FUNCTIONS

3.1 Design and Implementation Goals for the Controller

The primary goal in designing and implementing the controller subsystem of MDBS is to alleviate the controller limitation problem, i.e., to limit the amount of work that the controller must perform. The choice of a solution to the controller limitation problem is prompted by another design and implementation goal for MDBS, that of minimizing communication among the backends and between the backends and the controller. Without increasing workload and excessive communication, the throughput of MDBS will continue to increase as additional backends are added.

The controller limitation problem occurs in RDBM [Auer80], a relational database machine, where a general-purpose minicomputer is used to control the different hardware components of the system and to pre-process user requests. Request pre-processing includes a detailed analysis of the request to determine the pages in the secondary memory to be accessed. The speed of the minicomputer is therefore a limiting factor to the throughput of RDBM. Consider a simplified example where preprocessing a user request requires 10 seconds of CPU time at the minicomputer, regardless of the number of backends in the system. The throughput rate of RDBM is limited to 6 requests per minute.

Another view of the controller limitation problem is from the perspective of response time. The total response time of the system may be viewed as the sum of controller execution time and backend execution time. Adding more backends can decrease the backend execution time, but controller execution time remains constant. So in order to minimize request execution time, we must also minimize controller execution time.

Our controller design is based on the principle that the major portion of the MDBS workload should be distributed among the backends. In adherence to this principle, the controller is conceptually simple and includes primarily those functions which cannot be performed by the multiple backends.

3.2 The Concept of "Traffic Unit"

Input to MDBS originates from a user working at some host computer. The host computer translates the user´s instructions into the MDBS Data Manipulation Language (DML) and transmits the translated requests to MDBS. This transmission or "traffic" may take two forms: it may be a single request, or it may be a transaction. Recall that in MDBS terminology, a transaction is defined to be a pre-specified set of requests which the user may use repeatedly.

In order to generalize the description of input to MDBS, we introduce the concept of a traffic unit. A traffic unit may be a single request or a transaction. The identification of a traffic unit is important to the host, since it must return to the user all output from MDBS associated with the traffic unit. The recognition of a traffic unit as a single request or as a transaction is also important to MDBS, since transactions must be processed in a manner which preserves the consistency of the database. Since the traffic unit is recognized in the host, we assume that the host will associate with each traffic unit currently in the system a unique identifier, which we call the traffic id.

3.3 The Structure of the Controller

The MDBS Controller is implemented in three functional categories: Request Preparation, Insert Information Generation, and Post Processing. The Request Preparation functions include those which must be performed before a request or transaction can be broadcasted to the backends. The Insert Information Generation functions include those which must be performed during the processing of an insert request to furnish additional information required by the backends. The Post Processing functions include those which must be performed after replies are returned from the backends, but before results of a request or a transaction are forwarded to the host machine. These three categories of functions have been described in Chapter 2 of this report. In this chapter, we present details of the implementations of these functions.

3.3.1 The Request Preparation Functions

The Request Preparation functions include the Parser and Request Composer. The Parser function parses the requests and checks for syntax errors. The Request Composer function transforms a parsed request into the form required for processing at the backends. These functions have been described in Section 2.1.1. Here, we emphasize the implementation.

(A) The Parser Function

Parser does both lexical and syntactic analyses of the MDBS DML statements. Input to Parser is in terms of a traffic unit, i.e. either a single request or a group of requests which constitute a transaction. As described in Section 2.1.1, the various outputs of the parser are the error messages and aggregation operators to the Post Processing functions, and correctly parsed requests to the Request Composer function.

The lexical analyzer was built using the LEX program available with the UNIX operating system. LEX [Lesk79] 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 (i.e., the tokens of the MDBS DML statements) 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.

The parser was built using the YACC program available with UNIX. YACC [John79], "Yet Another Compiler-Compiler", was 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, i.e., 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 input DML statements are syntactically correct. For a detailed explanation of YACC, see [John79].

(B) The Request Composer Function

The Request Composer receives parsed requests from the Parser, and transforms them into the form required for processing at the backends. Recall from Section 2.1.1 that update requests with type-III and type-IV modifiers require Request Composer to generate a retrieve request, and a subsequent update request with a type-0 modifier. Request Composer also initates the actions required for the insertion of updated records that have changed cluster. Since the implementation of Request Composer is straightforward, it will not be described further.

3.3.2 The Insert Information Generation Functions

Insert Information Generation consists of three functions: Backend Selector, Cluster Id Generator, and Descriptor Id Generator. When processing an insert request, the Backend Selector function determines the backend at which the record is to be inserted. The Cluster Id Generator function produces new cluster ids for new clusters. The Descriptor Id Generator function produces new descriptor ids for new descriptors. The functions are described in Section 2.1.2. Before we describe any implementation details, let us review the types of descriptors which are defined in MDBS.

As described in Chapter 1, records in the database are clustered on the basis of attribute values and attribute value ranges called descriptors. There are three types of descriptors: type-A, type-B, and type-C. A type-A descriptor defines an inclusive range of values. Each type-A descriptor is a conjunction of a less-than-or-equal-to predicate and a greater-than-or-equal-to predicate. An example of a type-A descriptor is:

((SALARY >= 2,000) and (SALARY =< 10,000))

A type-B descriptor defines a single value. Each type-B descriptor consists of an equality predicate. An example of a type-B descriptor is:

(POSITION = Professor)

A type-C descriptor designates an attribute name as a type-C attribute. As records are inserted into the database, a single-valued descriptor is created for each unique value associated with the type-C attribute. These descriptors, which are identical to type-B descriptors, are referred to as type-C sub-descriptors.

Type-A and type-B descriptors, type-C attributes and type-C subdescriptors are created at database-load time. No additional descriptors can be defined after the database is loaded. Type-C sub-descriptors, however, will be created dynamically as new records are inserted into the database.

(A) The Backend Selector Function

In order to conform to the data placement strategy described in [Hsia81a], the controller must determine the backend at which the record is to be inserted. This is the function of Backend Selector.

The information required for selecting the backend is maintained in the cluster-id-to-next-backend table (CINBT). There is an entry in the table for each cluster. Each entry contains the number of the next backend into which records are to be inserted, and the remaining track capacity at that backend. The CINBT is created at database load time. CINBT is implemented as a data abstraction. The operations on this data abstraction, insert, find and update, will be invoked by Backend Selector in accessing CINBT.

At the end of the descriptor search phase in processing an insert request, each backend will send to Insert Information Generation the cluster id for the record to be inserted. Since the cluster-definition table (CDT) is not replicated, backends at which no records of a cluster are stored will not find a cluster id for that cluster. There is also the case where the record being inserted has caused a new type-C sub-descriptor to be generated; in this case, no backends will return a cluster id. When Backend Selector determines that all backends have responded, it will proceed to select the backend at which the record is to be inserted.

(B) The Cluster Id Generator Function

In order to save storage and time, each cluster is identified by a cluster id. The Cluster Id Generator generates a new cluster id when there is a new cluster. There are two cases which require a new cluster id. These cases are described in (A) above.

(C) The Descriptor Id Generator Function

When an insert request contains a record with a type-C attribute and the value associated with that attribute does not appear in a type-C sub-descriptor, a new type-C sub-descriptor will be created. The assignment of descriptor ids is handled by the Controller to prevent coincidental creation of different descriptor ids by the backends for the same descriptor. If two simultaneous insert requests requiring the creation of the same type-C sub-descriptor were processed by the backends independently, different descriptor ids would be assigned for the same descriptor. In MDBS, descriptors must have unique ids.

Descriptor Id Generator will generate a new descriptor id when requested, and broadcast descriptor id and descriptor to all backends. Descriptor Id Generator will retain a list of all descriptors to which it has assigned descriptor ids. This list will be consulted each time a request for a new descriptor id is received in order to prevent coincidental creation of different descriptor ids for the same descriptor. The list will also be purged periodically.

3.3.3 The Post Processing Functions

The Post Processing functions include Aggregate Post Operation and Reply

Monitor. The Aggregate Post Operation function performs the final aggregate operation on partial aggregate results returned from the backends. The Reply Monitor function collects all the results for a request or transaction, and forwards them to the host machine. These functions are described in Section 2.1.3. No further implementation details are presented here.

3.4 The Process Structure of the Controller

Since a message-oriented approach to concurrency control is being used, we must choose a process structure for the Controller. There are several obvious choices.

First, all of the functions of the Controller can be combined into one process. This alternative is unattractive because it limits the Controller to one function at a time. A greater degree of concurrency can be obtained by using multiple processes and the multiprogramming facilities of the underlying operating system. A second alternative is to create a process for each of the seven functions of the Controller. While this does allow a high degree of concurrency, it 'is unattractive because of the 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 above. There are three processes: the Request Preparation process, the Insert Information Generation process, and the Post Processing process. Look again at Figures 10, 11, 12 and 13 from Chapter 2. These figures show the flow of data between Controller and Backends functions for insert, delete, retrieve, and update requests. Requests flow from the host through the Request Preparation process to the Backends, and from the Backends through the Post Processing process to the host. In the case of insert and update requests, the Insert Information Generation process will be exchanging data with Directory Managment in the Backends. Notice that the only interprocess communication in the Controller will be between the Request Preparation and Post Processing processes. This is the organization we adopt for the process structure of the MDBS Controller.

4.0 AN IMPLEMENTATION OF BACKEND FUNCTIONS

As discussed in Section 3.1, a basic design goal of MDBS is to assign as much work as possible to the backends in order to alleviate the controller limitation problem. Consequently, the backends functions are more complex than those of the controller. The functions of the backends fall into three categories: Directory Management, Record Processing and Concurrency Control.

The Directory Management functions perform directory operations such as cluster determination, address generation, and directory table maintenance. According to the incremental development strategy described in Chapter 1, two versions of Directory Management will be developed. A simplified Directory Management, where all directory information is stored in main memory, is described in [Kerr82]. This simplified Directory Management will be used in the first three versions of MDBS (MDBS-IV and V will employ a secondary-memory-based directory management).

The Record Processing functions perform operations such as record selection and attribute value extraction. The design of these functions is described in detail in Section 4.1.

A second design goal for MDBS is that the software should support concurrent execution of requests in the backends in order to maximize system throughput. The cluster-based concurrency control functions, described in [Hsia81b], will be implemented in Version II of MDBS. In Section 4.2, we present a preliminary design of the Concurrency Control functions.

4.1 The Record Processing Functions

The Record Processing functions are: Physical Data Operation and Aggregate Operation. The Physical Data Operation function includes a control subfunction and a subfunction for each type of request. The Retrieve Processing Subfunction, the Insert Processing Subfunction, the Delete Processing Subfunction, and the Update Processing Subfunction are invoked by the Control Subfunction according to the type of request being processed. The Aggregate Operation function includes subfunctions which accumulate partial aggregate results for a request when an aggregate operation is specified for an attribute in the target-list. The Aggregate Operation Subfunctions are invoked as required by the Retrieve Processing subfunction.

The Retrieve Processing subfunction, the Insert Processing subfunction, the Delete Processing subfunction, and the Update Processing subfunction are described in detail in the sections which follow. Here, we give a general description of the Control subfunction.

The input to Record Processing comes from the Directory Management functions. Input data includes:

(1) a request;

(2) a set of physical (disk) addresses of the tracks which contain data relevant to the request;

(3) in the case of an insert request, an indicator which is used to determine whether the record is to be placed on a new track.

The specific form of the output varies with the type of request; a general description of the output is a signal to the Controller that execution of the request is completed, and the results of execution.

The sequence of events is as follows:

Step 1: Input is received from Directory Management.

Step 2: The proper subfunction is invoked according to the request type.

Step 3: The results are sent to the Controller.

Step 4: A completion signal is sent to the Controller.

The results of a retrieve or an update request may include many records. Thus, the results are buffered independently via a data abstraction, the Block_Buffer_Abstraction, which is also described below.

4.1.1 The Block_Buffer_Abstraction

In MDBS, a cluster may correspond to more than one physical track of data on the disk. Therefore, for one cluster, there may be more than one physical address in the set of addresses furnished to Record Processing by

Directory Management. Data are accessed from or to the disk track by track. So, a fixed-length buffer can be used for input data.

The amount of output data varies from request to request. This implies that, given a fixed-length output buffer, the Record Processing functions must include logic to empty the output buffer when it is filled during execution of a request. In order to simplify the Record Processing functions, a data abstraction is used to implement a virtual variable-length output buffer. This technique has two advantages. First, the Record Processing functions will not need to include logic to monitor the state of the output buffer. Second, all the logic required to use the communication interface for sending results to the Controller can be localized in the code of the data abstraction.

The Block_Buffer_Abstraction furnishes a data object, the Result_Buffer, and a set of operations. The operations include a function to reserve a buffer, a function to stuff data into a buffer, and a function to flush a partially filled buffer. The actual data structure used by the abstraction is a fixed-length buffer. However, the stuff operation includes logic to empty filled buffers. It appears to the user that the output buffer is as large as required.

4.1.2 The Retrieve Processing Subfunction

A retrieve request has the form:

RETRIEVE Query Target-List [BY clause] [WITH pointer]

The purpose of the Retrieve Processing is to fetch the clusters of relevant data from the disk, to select from the clusters of relevant data the records satisfying the query, and to output the results according to the target-list and the optional BY and WITH clauses.

The algorithm is as follows:

Step 1: Reserve a result buffer.

Step 2: For each address in the set of track addresses furnished by Directory Management, fetch the track from the disk into the track buffer in the main memory.

Step 3: Examine the records in the track buffer one-by-one. If a record is marked for deletion, disregard it. If a record does not satisfy the query of the request, disregard it. If a record satisfies the query of the request, extract the values for the attribute names in the target-list of the request; if an aggregate operation is specified for an attribute on the target-list, invoke the appropriate aggregation subfunction with the appropriate value. Stuff results from extraction and/or aggregation into the result buffer. Repeat for each record in the track buffer.

Step 4: Repeat steps 2 and 3 until the set of track addresses is exhausted.

Step 5: Flush the result buffer.

If the optional WITH clause is included, a pointer or physical address of the record is stuffed into the result buffer for each record. The optional BY clause is used in conjunction with an aggregate operator, as explained in the next section.

4.1.3 The Aggregation Subfunctions

MDBS supports five aggregate operations on attributes in the target-list of retrieve requests. These are AVG, SUM, COUNT, MAX and MIN. An example of a target-list is:

(DEPT, AVG(SALARY))

No aggregate operator is specified for the attribute DEPT; the values of DEPT will be retrieved from all records identified by the query. The aggregate operator AVG will be applied to the values of SALARY retrieved from all records identified by the query. Thus, the average salary will be obtained.

An optional BY clause may be used with an aggregate operator. Assume that we wish to find the average salary of employees in each department. This can be achieved by using a retrieve request with the target-list (AVG(SALARY)) and the clause BY DEPT.

The aggregation subfunctions are invoked by the Retrieve Processing subfunction as required. For AVG, a sum of values and a count is accumulated. For SUM, a sum of values is accumulated. For COUNT, a count of values is accumulated. For MAX and MIN, the maximum and minimum elements are selected.

4.1.4 The Insert Processing Subfunction

The insert request has the form:

INSERT Record

The purpose of the Insert Processing subfunction is to insert the record in the request into a cluster. The record may be added to a partially-filled track of data, or may be inserted as the first record of a newly allocated track. The input to Record Processing for an insert request includes a new-track indicator. Since only one record is being inserted into one track of one cluster, Directory Management will furnish only one track address.

The algorithm for the Insert Processing subfunction is very simple :

```
Step 1: If the new-track indicator is off (meaning that the record is
        to be added to a track that already contains other records
        from the cluster), then fetch the track from the disk into
        the track buffer. If the new-track indicator is on, then in-
        itialize the track buffer (no data are fetched from the
        disk).
Step 2: Insert the record in the request into the track buffer.
Step 3: Store the track buffer on the disk.
```

4.1.5 The Update Processing Subfunction

The update request has the form:

UPDATE Query Modifier

The modifier in an update request specifies the new value to be taken by the attribute being modified. The modifier may be one of the types described below.

Type-0 : <attribute = constant>

Type-I : <attribute = f(attribute)>

Type-II : <attribute = f(attributel)>

Type-III : <attribute = f(attribute1) of Query>

Type-IV : <attribute = f(attribute1) of Pointer>

The Update Processing subfunction handles requests with modifiers of type-0, I or II. An update request with the modifier of type-III or type-IV is decomposed by the Controller into a retrieve request followed by an update request of type-0.

The main function of Update Processing subfunction is to select records satisfying the query and to update the value of the attribute specified by the modifier. When a type-0 modifier is specified, the new value is the constant from the modifier. When a type-I modifier is specified, the new value is a function of the old value. When a type-II modifier is specified, the new value is a function of the value of some other attribute in the record.

If the attribute being updated is a directory attribute, the updated record may change cluster. This occurs when the updated value does not correspond to the same descriptors as the value before update. In this case, the set of descriptors which can be derived from the record is not the same as the set of descriptors which defines the current cluster. If the updated record changes cluster, then the original record is marked for deletion and the updated record is sent to Request Composer in the Controller. Request Composer will generate an insert request for the updated record. If the updated record does not change cluster, then it is simply rewritten in the same cluster.

The algorithm is as follows:

Step 1: Reserve a result buffer.

Step 2: For each address in the set of track addresses furnished by Directory Management, fetch the track from the disk into the track buffer in the main memory.

Step 3: Examine the records in the track buffer one-by-one. If a record is marked for deletion, disregard it. If a record does not satisfy the query of the request, disregard it. If a record satisfies the query of the request, compute the new value according to the modifier and update the record in the track buffer. Check the updated record to determine whether it changes cluster. If it does, then the updated record is added to the result buffer and marked for deletion from the track buffer.

Step 4: After all of the records in the track buffer have been examined, store the track buffer back to the disk.

Step 5: Repeat Step 2 through Step 4 until the set of track addresses is exhausted.

Step 6: Flush the result buffer and send the results to Request Composer in the controller.

4.1.6 The Delete Processing Subfunction

The delete request has the form:

DELETE Query

The purpose of the Delete Processing subfunction is to delete all the records satisfying the query. Records are not physically deleted from the database. They are instead marked for deletion. Records will be physically deleted only when the database is reorganized.

The algorithm is as follows:

Step 1: For each address in the set of track addresses furnished by Directory Management, fetch the track from the disk into the track buffer in the main memory.

Step 2: Examine the records in the track buffer one-by-one. If a record is marked for deletion, disregard it. If a record does not satisfy the query of the request, disregard it. If a record satisfies the query of the request, set a deletion flag in the record.

Step 3: Repeat Step 1 and Step 2 until the set of track addresses is exhausted.

Step 4: Store the track buffer on the disk.

4.2 Concurrency Control

In the previous sections, all consideration of the concurrent execution of requests has been omitted. However as was mentioned in Chapter 1, the backends must allow concurrent execution of requests in order to assure efficient processing of the requests. This section will first present a brief review of the concurrency control mechanism which was described in detail in [Hsia81b]. Then it will provide more details concerning the implementation.

Concurrency control is a mechanism by which we will insure the consistency of the database while allowing concurrent execution of multiple requests. To insure the consistency of the data, locks are utilized. These locks are administered at the cluster level (i.e., individual clusters are locked). There are five phases of execution of a request in the presence of access control and concurrency control. First, directory management determines the clusters needed by the request. Second, cluster access control determines the authorized clusters. Third, concurrency control determines when all clusters needed by the request are available. Fourth, address generation determines the record addresses. Finally, record processing actually executes the request.

4.2.1 Two Types of Consistency

The MDBS Concurrency Control mechanism differs from others in the types of locks as well as in their utilization. The mechanism distinguishes the four types of requests (Update, Retrieve, Insert, and Delete) and utilizes a different lock mode for each type.

There are two types of consistencies which must be assured. The first type of consistency is called inter-consistency. One example of the type of problem we are concerned with is two concurrent updates of a record, which might result in the loss of one of the updates. This problem must be considered in both single and multiple backend systems. To preserve inter-consistency, non-concurrent execution must be assured among requests which may have different results when executed simultaneously. Requests which may execute concurrently are called compatible requests. The compatibility of two requests depends on the mode of access, e.g., two retrieve requests are compatible whereas two update requests are not. When considering a new request, if the mode of the new request is not compatible with that of one of the earlier requests which is executing, then the execution of the new request must be delayed. Thus the MDBS concurrency mechanism locks clusters so that only compatible requests can be using a cluster at the same time.

As just described, requests are executed at the backends in the order they are received from the controller. Sometimes for performance reasons, however, it may be desirable to permute the order of execution of two requests that are not compatible. For example, suppose a sequence of three requests R1, R2 and R3 are received and R1 requires cluster C1, R2 requires clusters C1 and C2, while R3 requires cluster C2. In a single backend system, it would be possible to permute the execution of requests R2 and R3, allowing R3 to execute concurrently with R1 since R1 and R3 require different clusters. In order to permute the order of execution of requests in a multi-backend system, however, a mechanism must be found to assure that all backends execute the requests in the same order. Otherwise inconsistent results can again occur. Thus in a multi-backend system it is also necessary to assure _intra-consistency_, i.e., requests that are not compatible must execute in the same order at all backends.

A general mechanism to allow the permutation of requests that are not compatible would be complex because it would require communication among all the backends. However a simple mechanism can be found that will handle the special case involving an insert request. The actual insertion of a new record is performed at only one backend. It is not distributed across all the backends. Therefore, if the backends are allowed to permute a non-insert request and an insert request, then the _effective_ order of execution of the requests at all the backends is the order used by the backend which actually performs the insertion. In general, two requests that are not compatible are called _permutable_ if they do not have to be executed in the same order at all the backends. Thus we see that an insert request and a non-insert request are permutable and we can assure intra-consistency if we permute the execution order only of permutable requests.

The compatibility and permutability of requests can be summarized as follows:

	Delete	Insert	Update	Retrieve
Delete	C	P	N	N
Insert	P	C	P	P
Update	N	P	N	N
Retrieve	N	P	N	C

C = Compatible
P = Permutable
N = Not permutable and not compatible

This table shows that two delete requests, or two insert requests or two retrieve requests are compatible because they can be executed concurrently without the possibility of inconsistency developing. It also shows that an insert request can be permuted with a non-insert request, i.e., a delete, an update or a retrieve. As was explained above, this permutability of an insert request with a non-insert request is due to the fact that the actual insert occurs at only one backend. Only the delete, update or retrieve is actually performed at all the backends. Thus the effect is the same as it would have been if all the backends executed the requests in the order used by the backend performing the insert.

The concurrency control mechanism described in [Hsia81b] assures that requests which are not permutable or compatible are executed, without overlap, in the order received by the controller. Permutable requests can, however, be executed in any order so long as they do not overlap at the same backend. So as to keep track of all the requests, each backend maintains a queue of requests for each cluster, in the order in which the controller received the requests. Thus no later request can execute before an earlier request that is not permutable has been executed. In addition no permutable requests can execute concurrently, although the order of execution can be modified. On the other hand compatible requests can execute together.

4.2.2 Two Categories of Locks

Unfortunately, allowing the permutation of requests means that a new problem may now occur, the problem of starvation. It may be possible to permute one request indefinitely. Thus that request will never be allowed to execute. In order to prevent starvation, we introduce two categories of locks: "to-be-used" and "being-used". As soon as a request reaches a backend, it locks the clusters it needs in the "to-be-used" category. Before it can execute, it must convert the locks to the "being-used" category. Only requests which are locking a cluster in the "to-be-used" category are allowed to be permuted. Thus starvation can be prevented. Details of how this conversion of a lock from "to-be-used" to "being-used" and how this mechanism allows the permutation of requests while preventing starvation are discussed below. First however we must relate transactions to concurrency control.

4.2.3 The Notion of Transaction

A user may wish to treat a set of requests as a transaction. Such a set of requests is known by the user to preserve the consistency of the database if executed alone on a database system running on a single computer. Users may want execution of a transaction to begin before all the requests in the transaction have been provided to MDBS. In this case, we call the transaction incompletely-specified. Unfortunately, because all clusters required by the incompletely-specified transaction cannot be determined before execution of the transaction is to begin, there is no algorithm which allows the use of incompletely-specified transactions without sometimes having to backup one of two transactions which have been executing concurrently. Thus in MDBS, we have chosen to restrict transactions to those that are pre-specified, i.e., all the requests in a transaction must be submitted to MDBS at the same time and before execution of any of the requests in the transaction begins. Then MDBS must convert all locks to the "being-used" category before execution of the transaction can begin. Locks can then be released as requests in the transaction finish execution.

In the previous section, when we discussed compatible and permutable requests, we assumed the requests were not part of a transaction. We must now reexamine these concepts in the context of transactions. Since two compatible requests have no affect on each other, we can still allow their concurrent execution even when one is part of a transaction. On the other hand, the order of execution of two permutable requests does affect the result. Thus the whole transaction should be permuted, rather than one of its requests. Because of the complexity of permuting a whole transaction, we have chosen to permute only requests that are not part of a transaction.

4.2.4 Concurrency Control Using a Message-Oriented Approach

The concurrency control mechanism was described in [Hsia81b] using a procedure-oriented approach. Thus there was to be a lock table shared by all users. In addition, transactions were deactivated when a needed cluster was locked by other requests and were activated when the needed cluster became available.

This basic mechanism must now be transformed to reflect a message-oriented approach. In this approach, as described earlier, there is a concurrency control process. This process receives messages from the directory management process (a request to be executed) and from the record processing process (a report that a request has completed execution). When the concurrency control process determines that a request is ready for execution it forwards the request to record processing. The "shared lock-table" evident in the procedure-oriented approach now appears as a table internal to the concurrency control process. This table, called the cluster-to-traffic-unit table(CTUT), is described in Section (B) below. The concept of "deactivating" a transaction is replaced by having concurrency control hold the request in a queue until it can be forwarded to record processing for execution. The algorithms for concurrency control are described in Section (D) below.

(A) The Process Structure in the Backends

Once a message-oriented approach has been selected, it is necessary to break up the functions of each backend into processes. The most obvious choice would be to have one process per function, i.e., five processes corresponding to descriptor search, cluster search, concurrency control, address generation and record processing, respectively. (The sixth function, cluster access control is omitted because it is not included in our initial implementation.) However, since there is added overhead for each interprocess message, it is desirable to reduce the number of processes. One easy way to do this is to combine descriptor search and cluster search into a single directory management process. Address generation must take place after concurrency control, since records may be added to a cluster while a request is waiting to lock the cluster. Thus, address generation cannot be included in a directory management process. However, it could be combined with either concurrency control or record processing. For the purposes of discussing concurrency control, it is easiest to assume that address generation is not part of concurrency control. Thus the function of concurrency control is to schedule the execution of requests based on the clusters that are required as determined by directory management.

(B) Cluster-To-Traffic-Unit Table (CTUT)

As was described earlier, information about the locks held on each cluster is stored in the CTUT. This table contains a queue for each cluster. Each cluster queue contains an entry for each of the requests requiring that cluster. Each entry contains an identifier for the request (the traffic-unit and the request-number), the MODE of access required (delete, insert, retrieve or update), and the CATEGORY of lock held ("to-be-used" or "being-used"). A sample CTUT with four clusters is shown in Figure 14. This table contains entries for five single requests and one transaction consisting of two requests.

(C) Traffic-Unit-To-Cluster Table (TUCT)

In a procedure-oriented implementation there is a process associated with each user and this process keeps track of how many locks are still to be acquired before a transaction can be executed. However, in a message-oriented implementation, of course, there is no such process for a user. Thus this information must be maintained in a different way. The concurrency control process stores this information in a <u>traffic-unit-to-cluster</u>

Clusters	Traffic-Units		
C1	TU1 I BU	TU2 I BU	TU3 U BU
C2	TU4 I TBU	TU3 U BU	
C3	TU4 I BU	TU5,R1 D BU	
C4	TU5,R2 U TBU	TU6 I BU	

TU1 and TU2 are compatible and are executing. The lock for TU3 has been converted to "being-used", but since U and I are not compatible, TU3 must wait.

TU3 and TU4 have been permuted.

TU5,R1 would be permutable with TU4, except that it is part of a transaction.

C = Cluster

TU = Traffic-Unit
R = Request within traffic-unit

MODE of Request
D = Delete
I = Insert
R = Retrieve
U = Update

CATEGORY of Request
BU = Being-Used
TBU = To-Be-Used

Figure 14. A Sample of Cluster-To-Traffic-Unit Table (CTUT)

table (TUCT), which it can then use to determine the status of any traffic-unit. This table is essentially an inverse of the CTUT. It is a reference, by traffic-unit, of which clusters are required for each request of the traffic-unit. In addition, this table keeps track of how many requests of the transaction have not yet been sent to record processing for execution. Figure 15 shows the TUCT corresponding to the CTUT shown in Figure 14.

(D) The Processing of Concurrency Control Information

The concurrency control process receives messages from directory management and from record processing. A message from directory management consists of a new request to be executed and a list of clusters required by that request. A message from record processing means that execution of a request has been completed. Concurrency control must send messages to record processing notifying it to begin execution of a request.

In order to handle these messages, concurrency control must perform three basic functions. When a new traffic-unit is received from directory management, an initialization must be performed locking all the required clusters in the "to-be-used" category. When concurrency control receives a message from record processing that execution of a request has been completed, then concurrency control must remove that request from the TUCT (and CTUT) and determine the clusters that were locked by that request. Finally, whenever a new request is received or an old request has completed execution, concurrency control must try to convert as many locks in the clusters required by that request to the "being-used" category. When all locks required by a request have been converted to "being-used", the process must notify record processing to begin execution of the request.

```
 Traffic-     ||    Requests
   Units      ||
--------------++-----+-------------------
          TU1 || C1  |                  executing <---+
(one request) || BU  |                                |
--------------++-----+-------------------          compatible
          TU2 || C1  |                                |
(one request) || BU  |                  executing <---+
--------------++-----+----+--------------
          TU3 || C1   C2  |             waiting for C1 <---+
(one request) || BU   TBU |                                |
--------------++----------+--------------              permutable
          TU4 || C2   C3  |                                |
(one request) || BU   BU  |             executing <--------+
--------------++-----+----++-------------
          TU5 || C3  | C4  |            waiting for C3
(two requests)|| BU  | TBU |
--------------++-----+-----+-------------
          TU6 || C4  |                  waiting for C4
(one request) || BU  |
--------------++-----+------------------
```

TU	=	Traffic-Unit
C	=	Cluster
BU	=	Being-Used
TBU	=	To-Be-Used

* Note that a transaction must acquire all locks before it can proceed. It can, however, release the locks as each request finishes execution.

Figure 15. The Traffic-Unit-To-Cluster Table (TUCT) Corresponding to the CTUT in Figure 14

5.0 TESTING MDBS

In order to test MDBS two types of sample information must be made available. They are sample databases and sample requests. Therefore, a test of MDBS consists of loading a sample database and then executing one or more sample requests on the database.

5.1 The Need for the Generation of Test Databases and Lists of User Requests

In the first report [Kerr82], it was argued that a program to generate test databases would facilitate the testing process. A program, the Test File Generation Package, was developed for this purpose. It was also described in the first report.

A second program, the Test Request Generation and Execution Package, is being developed. This program is to assist in the generation of lists of sample requests to be executed and to facilitate the execution of the requests in a test session. In the following sections, we describe this Package.

5.2 The Generation of User Requests Lists

Several methods of generating lists of requests are possible. In addition, once the requests have been generated, several schemes for executing the requests are also possible.

5.2.1 User-Generated vs. Program-Generated Requests

As with the test files, a user may directly generate each request to be executed or a program may generate random requests based on some criteria chosen by the user. Since we anticipate our initial tests will use only a small number of requests and since we want to choose our requests to test certain features of MDBS, we have chosen to implement a package which first assists the user in the generation of short lists of requests and then facilitates the execution of lists of requests intended for certain features of MDBS. In other words, we are developing a package for user-generated test requests. Program-generated lists of requests are needed for performance evaluation experiments but are not needed for testing the features of MDBS. Such a package will be developed at a later date.

5.2.2 A Simple Test Package for a Single User

The first package developed is intended for testing Version I of MDBS, i.e., it assumes a single user wants to execute one request at a time. This package first assists the user in the generation of lists of requests. Once a list of requests has been generated it is saved in a file so that it can be executed at a later time. Thus the user does not have to type in sample requests repeatedly.

The test package works as follows. A user decides to have a <u>test session</u> consisting of several <u>test subsessions</u>. During each subsession the user can do one of the following:

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

(2) Generate a list of requests to be stored in a file for later use.

(3) Retrieve a list of requests that were previously stored in a file and then 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.

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

The user can continue with as many subsessions as desired. The user is also given a choice of two ways to examine the responses from MDBS. They may be displayed immediately at the user terminal and/or they may be saved in a file for later examination. The design of this package is given in Appendix D.

5.2.3 A Test Package for the Simulation of Multiple Concurrent Users

MDBS is, of course, designed to allow concurrent execution of requests by multiple users. Thus all versions of MDBS, except MDBS-I, must be tested with multiple concurrent users. In order to perform these tests there must be a way to simulate multiple users.

The simplest technique is to execute multiple copies of the package described in the previous section. Thus, if we had n copies of the package, we could simultaneously execute n different lists of requests - one for each concurrent user. Although easy to implement, there are two problems with this technique. First, setting up the n copies will be inconvenient, since we will either need n people sitting at different terminals or someone will have to run among a group of terminals. Second, replicating a test will be difficult. MDBS merges the requests as they are received from different users. The requests are then executed in this merged order, subject to alterations due to concurrency control restrictions. Even if in two tests the users all submit the same requests in the same order, there is no guarantee that MDBS will receive the combined requests in the same order. Thus the merged lists will be different and the two tests will not be identical. Although it will be possible to run the same sets of requests, it will be impossible to assure that MDBS will receive the requests in the same order. Thus MDBS will not be asked to perform exactly the same sequence of requests.

An alternative to running multiple copies of a package which can only simulate a single user is to run a new package that actually simulates multiple users. Such a package may be a modification of the single-user package. The main modification would be to associate each request with a particular user. Then the requests could be executed in turn thus simulating a multi-user system. We plan to use this approach for testing the later versions of MDBS.

5.2.4 A Test Package for the Generation of Random Requests

Like the Test File Generation Package, which generates test files with specified distributions of data values, this package generates certain types of requests for performance evaluation. Lists of requests vary in the mixture of the different request types they contain. Thus a user should be able to specify the percentage of RETRIEVE, INSERT, DELETE and UPDATE requests to be generated. In addition, each non-insert request has a query part. Some queries may be simple, say with one or two predicates. Others may be more complex, say with 10-15 predicates. Thus the user should also be able to specify the complexity of the requests being generated.

MDBS is likely to be more effective handling some forms of requests than others. Thus, it is desirable to perform experiments with different distributions of the request types. A package for the generation of random requests is to be developed for performance evaluation studies.

6.0 OUR SOFTWARE ENGINEERING EXPERIENCE

Well-known software engineering techniques have been applied to the development of application programs and the writing of compilers and operating systems. They have not, however, been widely applied to database system implementations. Our goal is not limited to the production of a prototype MDBS, but is aimed toward application of software engineering techniques to the development of the system. In the application, we are trying to identify the adequacy and applicability of the software engineering techniques used. We also attempt to modify the existing software engineering methodologies and propose new methodologies to tailor them for effective software engineering of database systems. In [Kerr82], we described the initial techniques that we were going to use. In this chapter, we describe some of the techniques that have been most effective. We also describe the new techniques that we have added to our initial techniques.

We conclude this report by giving the current status of the implementation.

6.1 The Effectiveness of the Techniques Used

Different software engineering techniques have been used in the development of MDBS. They include a modified chief-programmer team organization, uniform documentation standards, a formal system specification language, use of structured walkthroughs, incremental development, top-down design strategy combined with the use of data and service abstractions, structured coding and a testing approach. Our finding is that most of the techniques may be used in prototyping the database system, i.e., MDBS. In this section, we describe the techniques that have been most effective.

6.1.1 The Use of Structured Walkthroughs

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.

We have been using this technique at both the design stage and the coding stage. All detailed program specifications and source codes are reviewed by walkthrough committees. The status of a task can be determined by reviewing the walkthrough reports for that task. Figure 16 shows a sample walkthrough report. A good reference describing the structured walkthrough technique is [Your79a].

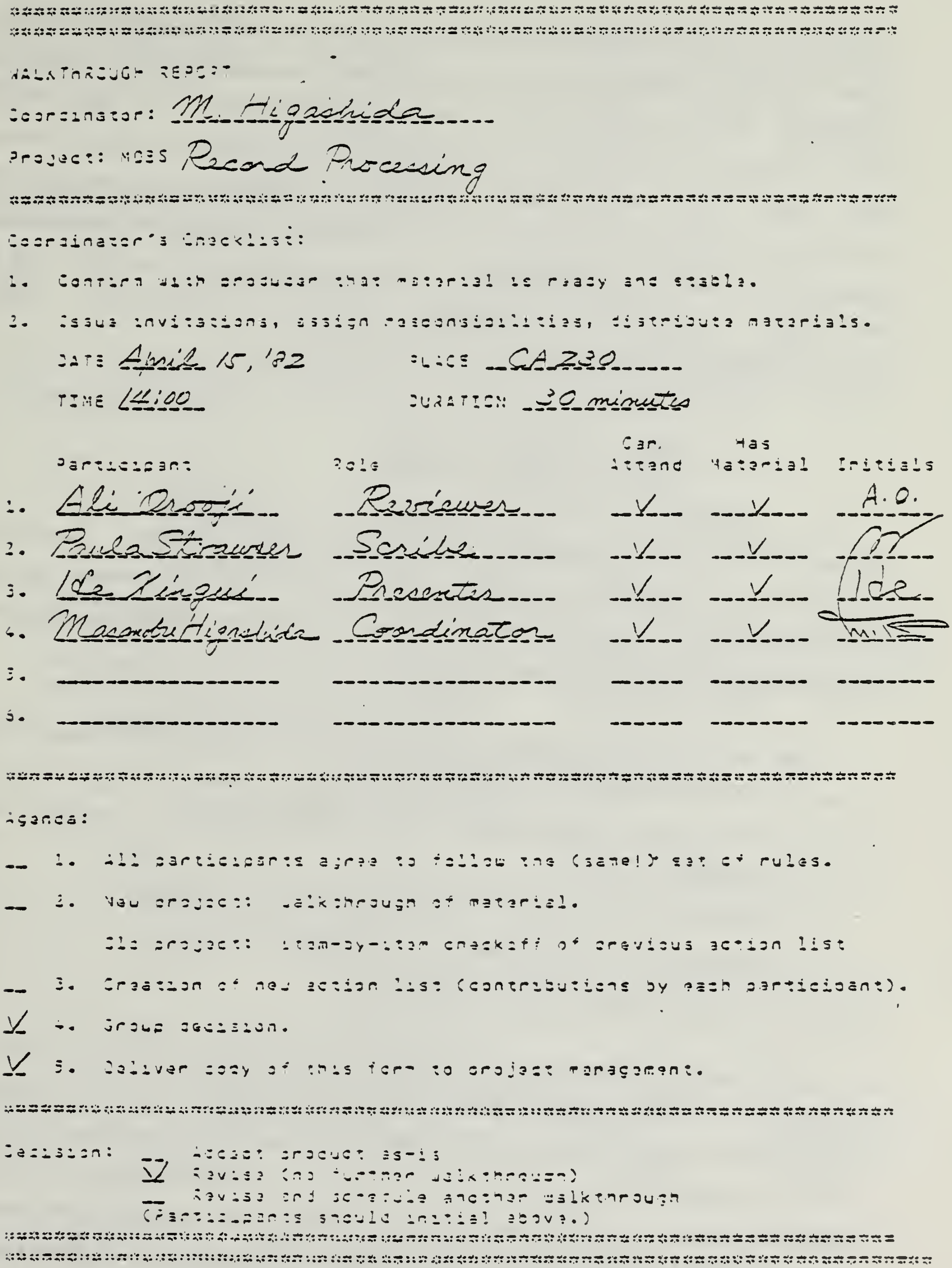
The use of structured walkthroughs has helped us to identify most of the design and program code problems. Furthermore, most of the suggestions made by the reviewers in the walkthroughs have been very useful to the presenters. A presenter, of course, investigates the comments and suggestions made about his work, instead of simply modifying his work to incorporate the suggestions.

6.1.2 The Use of a Formal Systems Specification Language (SSL)

Our original design methodology was a systems specification language (SSL) modeled on the process design language (PDL) described in [Ling79]. The original SSL is described in [Kerr82]. The SSL which we now use is based on our original SSL and it is intended to describe systems of any size. The current SSL is characterized by a number of constructs for the expression of the different levels of a system: system, subsystem, module and procedure. A <u>system</u> is at the highest level of the hierarchy. MDBS, for example, is a system.

At the second highest level of the hierarchy, we have the level of subsystem. A <u>subsystem</u> 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 <u>module</u> is



WALKTHROUGH REPORT

Coordinator: M. Higashida

Project: MCES Record Processing

Coordinator's Checklist:

1. Confirm with producer that material is ready and stable.
2. Issue invitations, assign responsibilities, distribute materials.

DATE April 15, '82 PLACE CA230

TIME 14:00 DURATION 30 minutes

	Participant	Role	Can Attend	Has Material	Initials
1.	Ali Orooji	Reviewer	✓	✓	A.O.
2.	Paula Strawser	Scribe	✓	✓	PS
3.	Ide Xingui	Presenter	✓	✓	Ide
4.	Masaoki Higashida	Coordinator	✓	✓	
5.					
6.					

Agenda:

□ 1. All participants agree to follow the (same!) set of rules.

□ 2. New project: walkthrough of material.

Old project: item-by-item checkoff of previous action list

□ 3. Creation of new action list (contributions by each participant).

☑ 4. Group decision.

☑ 5. 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 16. A Sample Walkthrough Report

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. 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
endif
```

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 17 shows a typical SSL procedure specification.

The use of a formal system specification language has been very effective. More specifically, by using the SSL:

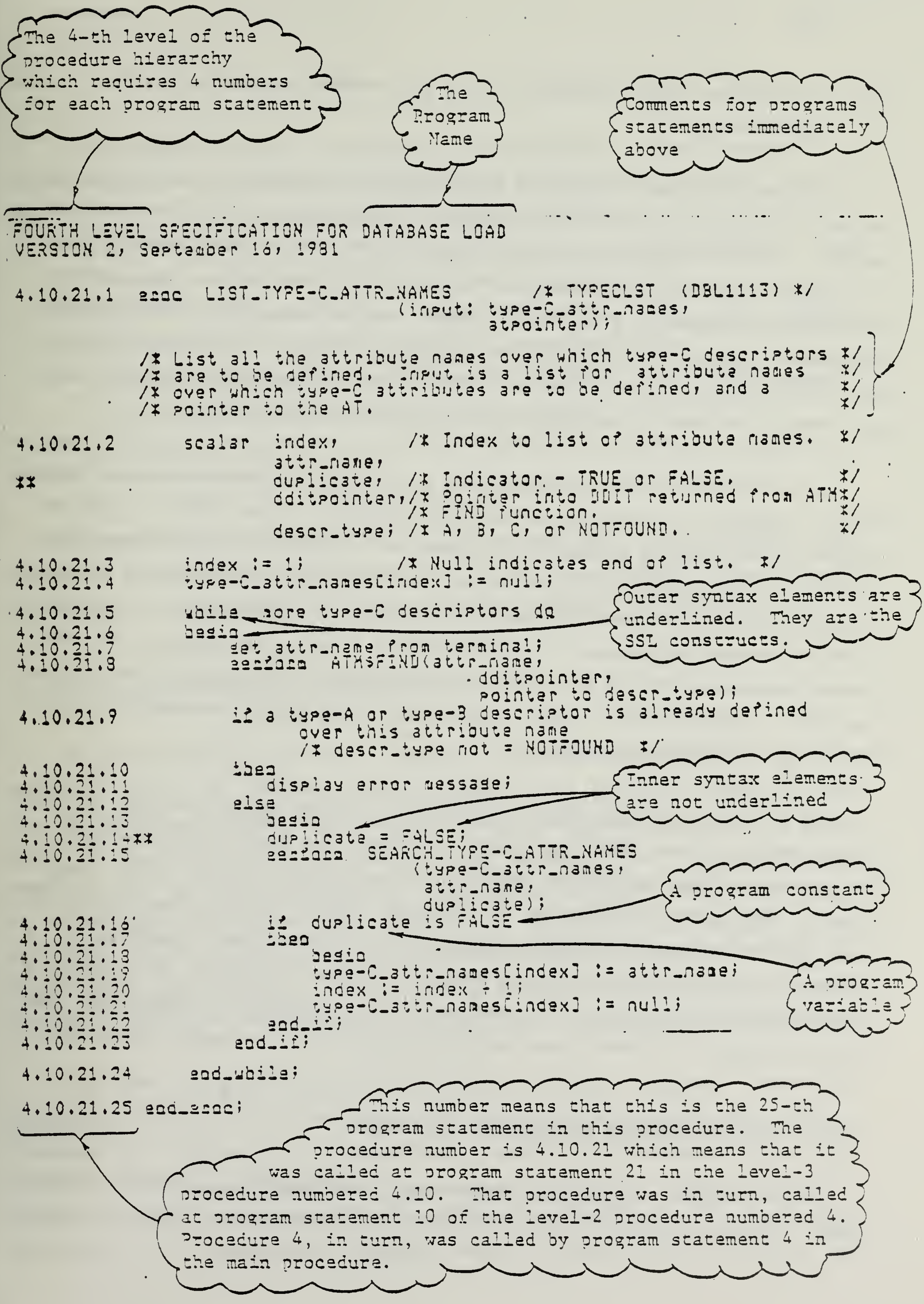
(1) Precise and unambiguous communication among the project members is achieved.

(2) Complete and accurate documentations are produced.

(3) Dependence on individuals is reduced.

(4) Project management is easier.

One useful concept that we have not employed is multi-level data abstractions (having higher level data abstractions which use lower level data abstractions which in turn use lower level data abstractions, and so on). We have used data abstractions only at the lower levels. The reason for this is probably that we are not used to the concept of multi-level data abstractions. This concept, however, leads to better and well-structured design. Thus, it should be employed.



```
FOURTH LEVEL SPECIFICATION FOR DATABASE LOAD
VERSION 2, September 16, 1981

4.10.21.1   proc  LIST_TYPE-C_ATTR_NAMES          /* TYPECLST  (DBL1113) */
                              (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*/
                                     /* FIND function.                     */
                        descr_type; /* A, B, C, or NOTFOUND.              */

4.10.21.3       index := 1;         /* Null indicates end of list.  */
4.10.21.4       type-C_attr_names[index] := null;
4.10.21.5       while  more type-C descriptors do
4.10.21.6       begin
4.10.21.7          get attr_name from terminal;
4.10.21.8          perform  ATM$FIND(attr_name,
                                         dditpointer,
                                         pointer to descr_type);
4.10.21.9          if a type-A or type-B descriptor is already defined
                        over this attribute name
                        /* descr_type not = NOTFOUND  */
4.10.21.10         then
4.10.21.11            display error message;
4.10.21.12         else
4.10.21.13            begin
4.10.21.14**          duplicate = FALSE;
4.10.21.15            perform  SEARCH_TYPE-C_ATTR_NAMES
                                 (type-C_attr_names,
                                  attr_name,
                                  duplicate);
4.10.21.16            if  duplicate is FALSE
4.10.21.17            then
4.10.21.18               begin
4.10.21.19               type-C_attr_names[index] := attr_name;
4.10.21.20               index := index + 1;
4.10.21.21               type-C_attr_names[index] := null;
4.10.21.22            end_if;
4.10.21.23         end_if;
4.10.21.24      end_while;
4.10.21.25 end_proc;
```

The 4-th level of the procedure hierarchy which requires 4 numbers for each program statement
The Program Name
Comments for programs statements immediately above
Outer syntax elements are underlined. They are the SSL constructs.
Inner syntax elements are not underlined
A program constant
A program variable
This number means that this is the 25-th 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 17. A SSL Specification of a Program Procedure

6.1.3 A Top-Down Design Strategy and the Use of Data and Service Abstractions

A top-down design strategy is a natural choice for MDBS. The design and analysis study in [Hsia81a] and [Hsia81b] clearly describes the top level of 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, as depicted in Figure 2. Directory management, an example of a functional unit, includes the descriptor search, cluster search, and address generation phases of request execution.

At a lower level, one concept, data and service abstractions, is used which originated in the bottom-up design approach. Since MDBS is being developed as a prototype system and is aimed 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.

We did not, unintentionally, follow the top-down design strategy in the development of the MDBS controller. Instead, the functional components of the controller such as Request Composer and Reply Monitor were first determined. Then, the categories of functions such as Request Preparation and Post Processing were determined based on the functional components. A top-down design of the controller would have first identified the categories of functions such as Request Preparation, Insert Information Generation and Post Processing. It would have then decomposed them into smaller components.

6.2 Trying New Software Engineering Techniques

We have added new software engineering techniques to our initial techniques. In this section, we describe these new techniques.

6.2.1 The Use of Jackson Charts

Our original designs were developed using only SSL. More recently, we have begun using a technique, Jackson chart [Jack75], to represent the program structures. Three constructs are used in a chart:

(1) Sequence - Figure 18a shows a sample sequence. In this example, the sequence A consists of B followed by C followed by D.

(2) Iteration - Figure 18b shows a sample iteration. In this example, the iteration A consists of multiple occurrences of B.

(3) Selection - Figure 18c shows a sample selection. In this example, the selection A consists of one of B, C or D.

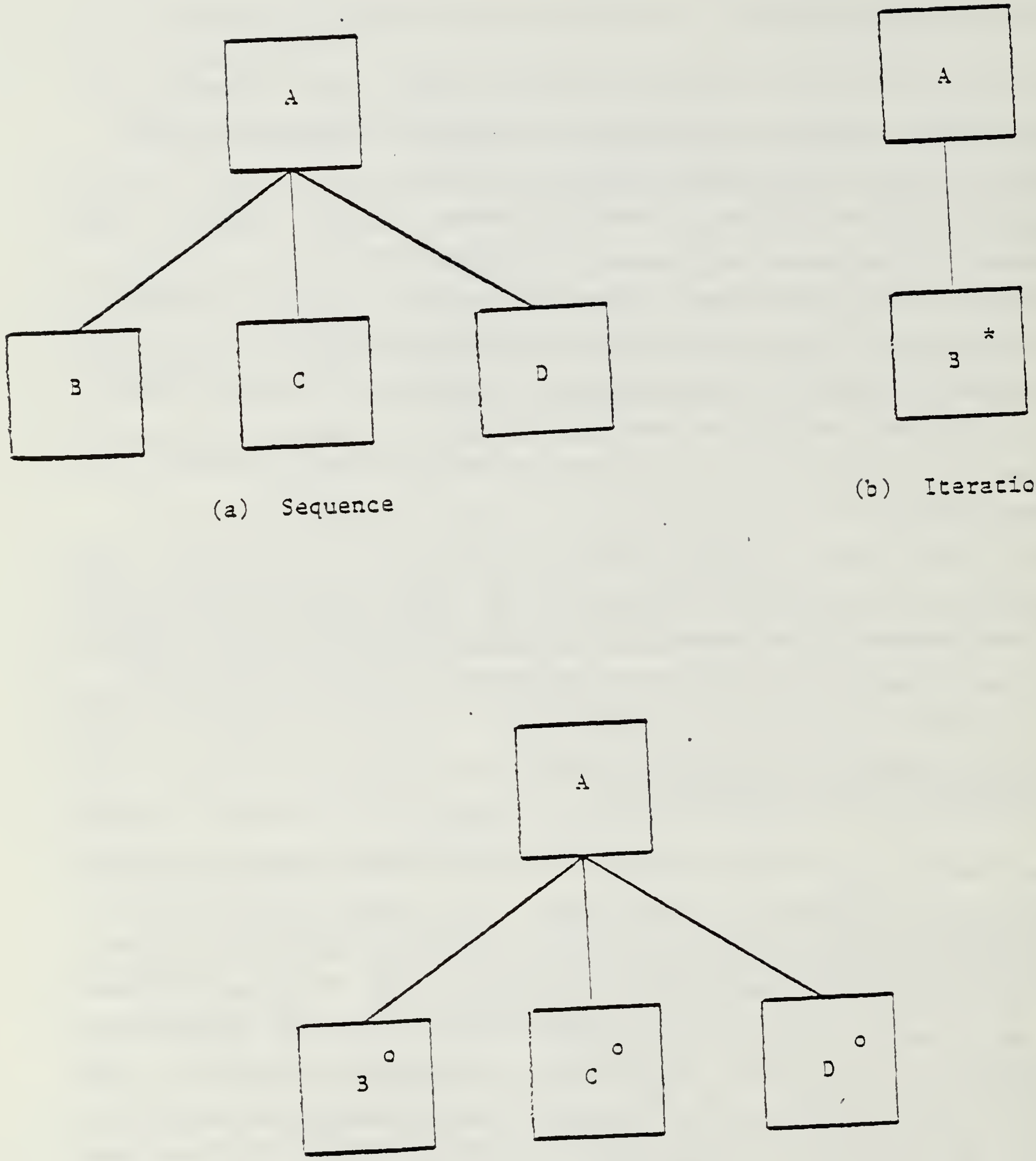
A sample program structure and its corresponding SSL are shown in Figures 19 and 20, respectively.

Jackson charts contain fewer details than the SSL specifications, and provide a two-dimensional representation of program structure. These charts, along with the SSL specification, are used to document the detailed design.

6.2.2 Standards for Module Decomposition

As explained in the previous chapters, the entire MDBS system has been designed as a set of discrete functional units. We propose to apply the same idea of functional decomposition at the level of subsystem design. We need some way to evaluate the modularity of our decomposition. This need became apparent when we began designing the top-level scheme for MDBS subsystems. It is necessary that we develop a unified view of the overall function of the subsystems of the controller and the backends before proceeding to design the abstractions and procedures. We have added to our collection of software engineering strategies two measures of modularity, or functional decomposition.

The first of these measures is strength of module cohesion [Your79b]. Cohesion is defined as the relatedness of processing elements within a single component of a system, i.e., a subsystem, a module or a procedure. The degree of relatedness determines the level of cohesion. Several levels of cohesion, ranked from least to most desirable, are recognized. A component is said to be functionally cohesive, most desirable level of cohesion, if



(a) Sequence

(b) Iteration

(c) Selection

Figure 18. The Constructs Used in a Jackson Chart

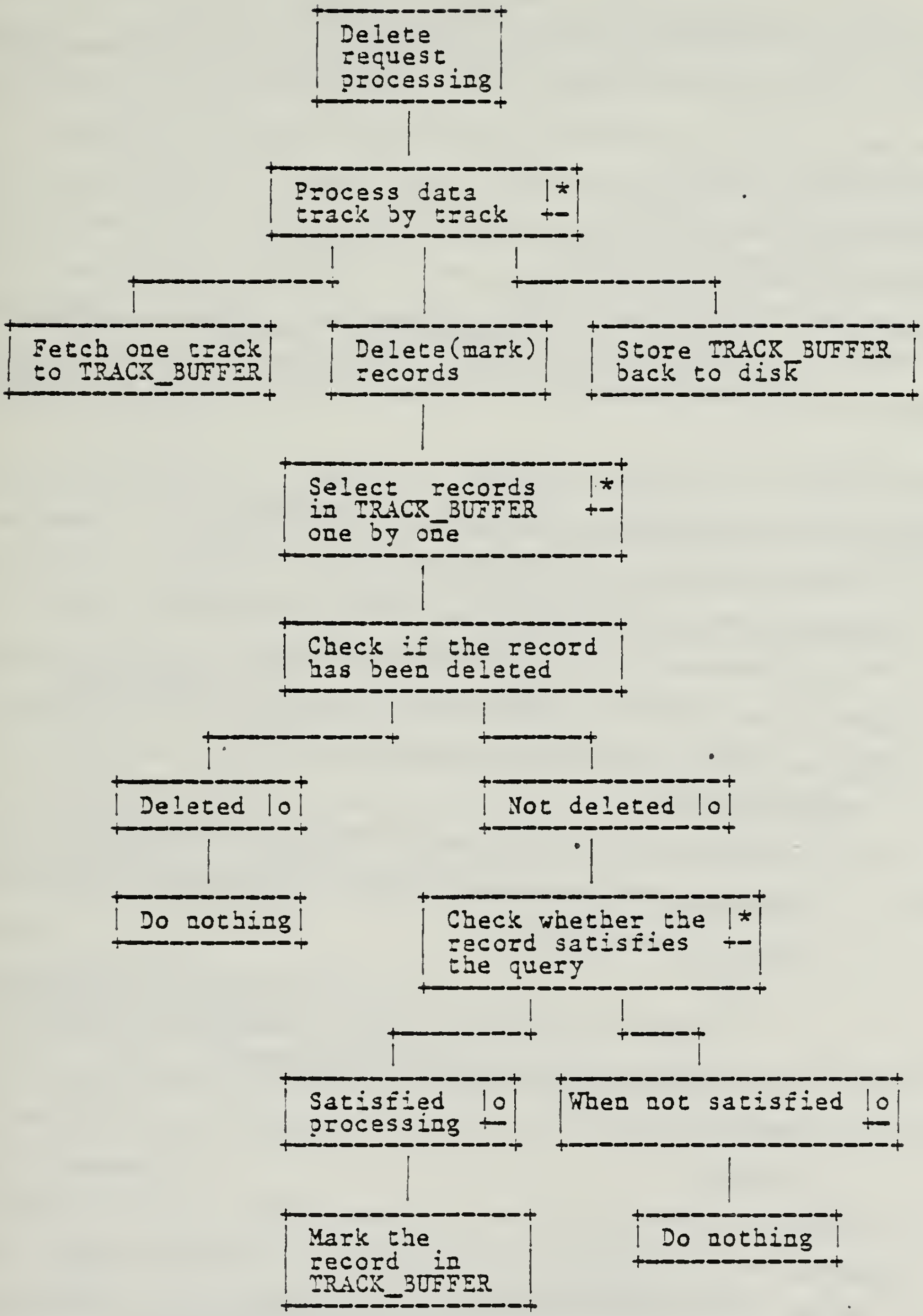

Delete request processing
Process data track by track *
Fetch one track to TRACK_BUFFER
Delete(mark) records
Store TRACK_BUFFER back to disk
Select records in TRACK_BUFFER one by one *
Check if the record has been deleted
Deleted o
Not deleted o
Do nothing
Check whether the record satisfies the query *
Satisfied processing o
When not satisfied o
Mark the record in TRACK_BUFFER
Do nothing

Figure 19. A Sample Program Structure

```
10.1  proc DELETE_PROCESSING( input: QUERY,ADDRESSES);
      /* This procedure is to be used for processing of DELETE requests. */

10.2    list QUERY: string;
10.3    set ADDRESSES: 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 .                                    */
10.7    for each address ADDRESSES(indexA) in ADDRESSES do
          /* Fetch one track into TRACK_BUFFER.                               */
10.8      perform FETCH_TO_TRACK_BUFFER(indexA,TRACK_BUFFER);
          /* Select records in TRACK_BUFFER one by one.                       */
10.9      for each record TRACK_BUFFER(indexB) in TRACK_BUFFER do
10.10       if the record is not marked for deletion
10.11         then begin
                /* Check whether the record satisfies the QUERY.  */
10.12           perform CHECK_QUERY(QUERY,TRACK_BUFFER,indexB,satisfied);
10.13           if satisfied='true'
                  then
                    /* Mark the retrieved record in TRACK_BUFFER(indexB).*/
10.14               perform DELETE(TRACK_BUFFER,indexB);
10.15           end if
10.16         end begin
10.17       end if
10.18     end for  /* indexB */
          /* Store TRACK_BUFFER back to disk.                  */
10.19     perform STORE_TRACK_BUFFER(indexA,TRACK_BUFFER);
10.20   end for     /* indexA */
10.21 end proc
```

Figure 20. The SSL Corresponding to the Sample Program Structure in Figure 8

"every element of processing is an integral part of, and is essential to, the performance of a single function". An ad hoc measurement is that the description for a functionally cohesive component should consist of one imperative sentence containing one transitive verb and one non-plural object. We have applied this ad hoc measurement to our current design work. The designer is required to write a functional description of each program component, say an abstraction, of the design. Each description begins with a single imperative sentence which concisely describes the function of that component. For example, the following sentence describes the function of a procedure of the MDBS controller, the Aggregate Post Operation

> Aggregate Post Operation performs the final aggregation operation on the partial aggregate results returned from the backends.

The second measure of modularity is the degree of interconnections between components. Coupling is a measure of the strength of interconnection between components. Several categories of coupling, ranked from lowest (best case) to tightest (worse case), are recognized. Two components are said to have no direct coupling, lowest coupling (best case), if each can function without knowledge of the other. We now give an example to show how we employed these standards for module decomposition. The original design for the controller, Figure 21, had a function called Insert/Update Information Generator. This function was intended to perform the following operations:

(1) to select a backend for record insertion when executing an insert request

(2) to generate new descriptor ids

(3) to generate new cluster ids.

In addition to performing the above operations by itself, Insert/Update Information Generator was intended to perform the following operations together with Request Composer:

(1) to initiate the actions required for the insertion of the records that change cluster as a result of executing an update request

(2) to generate update requests with type-0 modifier for update requests with type-III or type-IV modifier.

This function, Insert/Update Information Generator, is not functionally cohesive and it is highly coupled with Request Composer. Thus, we changed the original design of the controller.

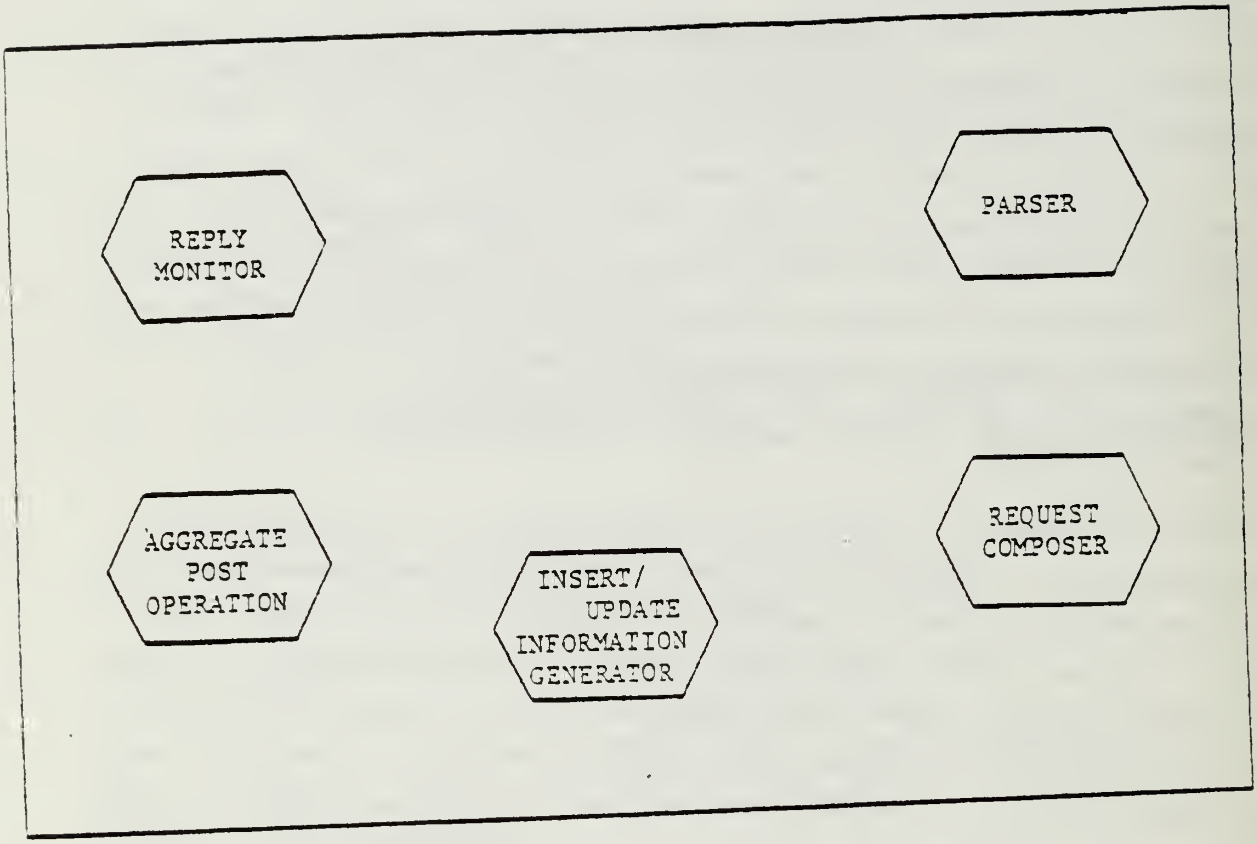


Figure 21. The Original Design for MDBS Controller

6.3 Current Status of the Implementation

The implementation of MDBS is underway. The detailed design of the controller is finished and coding has begun. The parser has been coded and tested. The first version of directory management is complete. The design for the second version has begun. The detailed design of record processing is finished and coding has begun. The design of concurrency control has begun. In addition to MDBS itself, we have completed several required auxiliary programs such as a database load utility. We expect to begin testing MDBS-I soon. We expect to finish MDBS-II, III and IV by the end of this year.

REFRENCES

[Auer80] Auer, H., "RDBM - A Relational Database Machine", Technical Report No. 8005, University of Braunschweig, June, 1980.

[Hsia70] Hsiao, D.K. and Harary, F.A., "A Formal System for Information Retrieval from Files," *Communications of the ACM*, Vol. 13, No. 2, February 1970; Corrigenda, *Communications of the ACM*, 13, 3, March 1970.

[Hsia81a] Hsiao, D.K. and Menon, M.J., "Design and Analysis of a Multi-Backend Database System for Performance Improvement, Functionality Expansion and Capacity Growth (Part I)", Technical Report, OSU-CISRC-TR-81-7, The Ohio State University, Columbus, Ohio, July 1981.

[Hsia81b] Hsiao, D.K. and Menon, M.J., "Design and Analysis of a Multi-Backend Database System for performance Improvement, Functionality Expansion and Capacity Growth (Part II)", Technical Report, OSU-CISRC-TR-81-8, The Ohio State University, Columbus, Ohio, August 1981.

[Jack75] Jackson, M.A., *Principles of Program Design*, Academic Press, 1975.

[John79] Johnson, Steven C., "Yacc: Yet Another Compiler-Compiler", *UNIX TIME-SHARING SYSTEM: UNIX PROGRAMMER'S MANUAL*, Bell Telephone Laboratories, Incorporated, Murray Hill, N.J., 1979.

[Kerr82] Kerr, D.S., et al., "The Implementation of a Multi-Backend Database Sysyem (MDBS): Part I - Software Engineering Strategies and Efforts Towards a Prototype MDBS", Technical Report, OSU-CISRC-TR-82-1, The Ohio State University, Columbus, Ohio, January 1982.

[Laue79] Lauer, H. and Needham, R., "On the Duality of Operating System Structures," in *Proc. Second International Symposium on Operating Systems*, IRIA, October 1978, reprinted in *Operating Systems Review*, Vol. 13, No. 2, April 1979, pp. 3-19.

[Lesk79] Lesk, M.E. and E. Schmidt, "lex - A Lexical Analyzer Generator", UNIX TIME-SHARING SYSTEM: UNIX PROGRAMMER´S MANUAL, Bell Telephone Laboratories, Incorporated, Murray Hill, N.J., 1979.

[Ling79] Linger, R.C., Mills, H.D., and Witt, B.I., Structured Programming - Theory and Practice, Addison-Wesley, 1979.

[Ston81] Stonebraker, M., "Operating System Support for Database Management," Communicatons of the ACM, Vol. 24, No. 7, July 1981, pp. 412-418.

[Your79a] Yourdon, E., Structured Walkthroughs (2nd Edition), Prentice-Hall, 1979.

[Your79b] Yourdon, E. and Constantine, L.L., Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, Prentice-Hall, 1979.

APPENDIX A
HOW TO READ AND FOLLOW THE PROGRAM SPECIFICATIONS

In Appendices B, C and D, a large number of MDBS programs are described and specified. These programs represent those parts of MDBS that have been designed since the first report in this series was written.

A.1 Parts within an Appendix

Each appendix begins with an introduction which outlines the major components of the design. For example, the design of the test request generation and execution package, presented in Appendix D, consists of two major components: one to generate lists of requests to be executed and then to execute those requests, the other to handle the output resulting from the execution of the requests. Accordinngly, each major component is described and specified in a separate part of the appendix. Thus Appendix D has Part I and Part II.

A.2 The Format of a Part

In each part, we provide the following documentation elements:

(1) Title of the part,

(2) Name of the design,

(3) Name of the designer,

(4) Date the design was first submitted,

(5) Dates of design modifications,

(6) Statements of the design purpose, and of the input and output requirements,

(7) Formal specifications of the input and output, if necessary,

(8) Procedure names used in the design,

(9) Jackson chart of the design,

(10) Data structures used in the design,

(11) Program specification of the design.

A.3 Documentation Techniques for a Part

In the previous section, we listed the various documentation elements. They are used to describe a design. Documentation elements 1 through 5 are written in English phrases. Document element 6 is written in prose. On the other hand, document elements 7 through 11 can be expressed more effectively using other means. Specifically, we use Backus-Naur form (BNF) for writing the specifications in document element 7.

The procedure names of document element 8 are shown in a program hierarchy. The use of the hierarchy makes clear the calling sequences of the procedures named. A Jackson chart as described in Section 6.2.1 and depicted in Figure 19 appears as element 9. The data structures of documentation element 10 are specified in either SSL or in the C programming language. In documentation element 11, the procedures, themselves, are specified in SSL.

Except for the programming team that writes the procedures, other teams will usually not be interested in the internal logic of the procedures. Consequently, they need only know the higher-level specifications of the procedures. SSL as described in Section 6.1.2 and depicted in Figure 17 is an ideal specification language for revealing the design of the procedures from a top-to-bottom-and-layer-to-layer way. It also works well with the hierarchical organization of procedures.

APPENDIX B

THE SSL SPECIFICATION FOR MDBS CONTROLLER

The system specification for the controller is given in this appendix. The specification consists of three parts, one for each process (task) in the controller.

In Part I, the Request Preparation process is specified. Insert Information Generation process and Post Processing process are specified in Parts II and III, respectively.

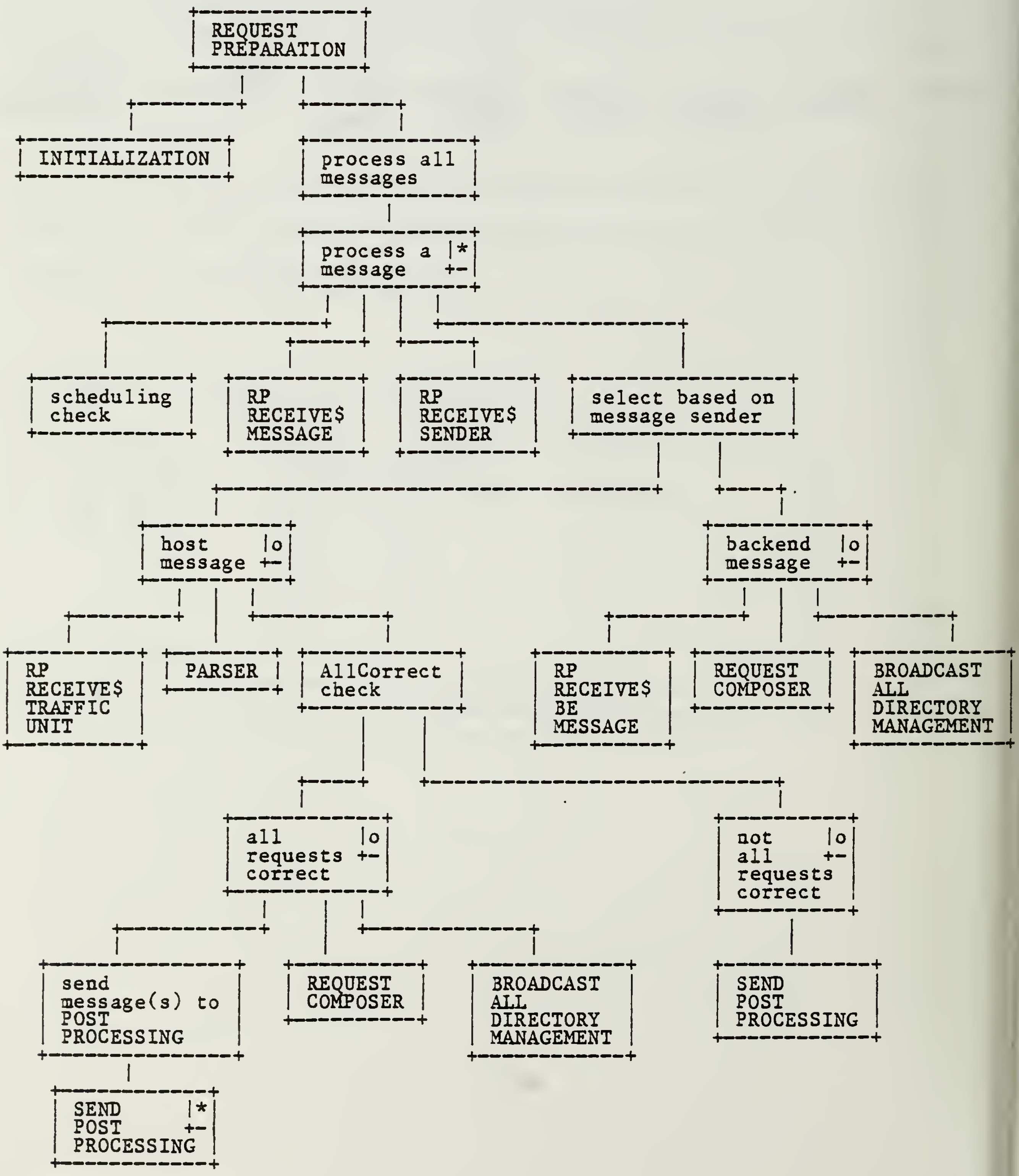
B.1 Part I - The Request Preparation Process

```
/*   (1) Part I     : The Request Preparation Process           */
/*   (2) Design     : REQUEST-PREPARATION                       */
/*   (3) Designers : A. Orooji,  Z.Z. Shi,  P.R..Strawser       */
/*   (4) Date       : Feb. 4, 1982                              */
/*   (5) Modified   : Feb. 19, 1982                             */
/*                    Apr. 12, 1982                             */
/*                    Apr. 26, 1982                             */
/*                    May  13, 1982                             */
/*                    May  19, 1982                             */
/*   (6) Purpose    :                                           */
/*      This is the Request Preparation process.  It consists of */
/* the functions which must be performed before a request or a   */
/* transaction can be broadcasted to the backends.               */
```

(8) Procedure Hierarchy for REQUEST-PREPARATION

```
                          REQUEST-PREPARATION
                                   |
  +-------+-------+---------+---------+---------+----------+------------+
  |       |       |         |         |         |          |            |
PARSER  REQUEST  RP        RP        RP        RP         SEND         BROADCAST
        COMPOSER RECEIVE$  RECEIVE$  RECEIVE$  RECEIVE$   POST         ALL
                 MESSAGE   SENDER    TRAFFIC   BE         PROCESSING   DIRECTORY
                                     UNIT      MESSAGE                 MANAGEMENT
```

(9) Program Structure of REQUEST-PREPARATION (Jackson Chart)



(10) Data Structures

```
/*   The data structure definitions are included in the program */
/* specifications.                                              */
```

(11) Program Specifications

```
task REQUEST-PREPARATION
    /* TrafficUnit :                                                  */
    /*    is either a request or a transaction                        */
    /* traffic-id :                                                   */
    /*    an integer that identifies a traffic unit                   */
    /* TrafficUnitPtr :                                               */
    /*    a pointer to (traffic-id,TrafficUnit)                       */
    /* AllCorrect :                                                   */
    /*    indicates whether or not all the requests in the traffic    */
    /*    unit are syntactically correct                              */
    /* request-id :                                                   */
    /*    consists of (traffic-id,request-no) which uniquely          */
    /*    identifies each request being processed by MDBS             */
    /* ParsedRequestsPtr :                                            */
    /*    is one of the following two                                 */
    /*       (1) a pointer to ((request-id,routing-indicator,         */
    /*           no-pred, parsed-request)[,...])  if all the requests*/
    /*           in the traffic unit are syntactically correct.       */
    /*       (2) a pointer to ((request-id,request,                   */
    /*           error-message)[,...])  if one (or more) of the       */
    /*           requests in the traffic unit is not syntactically    */
    /*           correct.                                             */
    /* FormattedRequestsPtr :                                         */
    /*    a pointer to ((request-id,routing-indicator,no-pred,        */
    /*    sched-no,formatted-request)[,...])                          */
    /* AggregateOperatorsMessagePtr :                                 */
    /*    a pointer to ((request-id,(attribute,                       */
    /*    aggregate-operator)[,...])[,...])                           */
    /* RequestCountMessagePtr :                                       */
    /*    a pointer to (traffic-id,request-count)                     */
    /* request-count :                                                */
    /*    number of requests in a traffic nuit                        */
    /* MessageType :                                                  */
    /*    indicates the type of a message                             */
    /* MessagePtr :                                                   */
    /*    a pointer to a message                                      */

    do initialization work;
    while ´true´ do     /* do forever */
       if according to the task scheduling this task should
                                         release the processor
          then
             release the processor and wait;
       end if
    /* get the next message for REQUEST-PREPARATION */
```

```
8.     perform RP-RECEIVE$MESSAGE;
       /* get the sender name of the next message for REQUEST- */
       /* PREPARATION                                          */
9.     perform RP-RECEIVE$SENDER(sender);
10.    case sender value
11.       'host machine':
             /* get the traffic unit */
12.          perform RP-RECEIVE$TRAFFIC-UNIT(TrafficUnitPtr);
             /* parse all the requests in the traffic unit */
13.          perform PARSER(TrafficUnitPtr,AllCorrect,ParsedRequestsPtr,
                     AggregateOperatorsMessagePtr,RequestCountMessagePtr);
14.          if AllCorrect
15.             then
                   /* all the requests in the traffic unit are */
                   /* syntactically correct                    */
                   /* send number of requests in the traffic   */
                   /* unit to POST-PROCESSING                  */
16.                MessageType := 'number-of-requests-in-a-
                                         traffic-unit message';
17.                perform SEND-POST-PROCESSING(MessageType,
                                                RequestCountMessagePtr);
18.                if there are aggregate operators
19.                   then
                         /* send the aggregate operators in the */
                         /* requests to POST-PROCESSING         */
20.                      MessageType := 'aggregate-operators message';
21.                      perform SEND-POST-PROCESSING(MessageType,
                                       AggregateOperatorsMessagePtr);
22.                end if
                   /* transform the requests into the form required */
                   /* for processing at the backends                */
23.                MessageType := 'PARSER message';
24.                perform REQUEST-COMPOSER(MessageType,
                           ParsedRequestsPtr,FormattedRequestsPtr);
                   /* send the requests to all the backends */
25.                perform BROADCAST-ALL-DIRECTORY-MANAGEMENT(
                                             FormattedRequestsPtr);
26.             else
                   /* one (or more) of the requests in the traffic */
                   /* unit has errors                              */
                   /* send the requests along with error messages  */
                   /* to POST-PROCESSING                           */
27.                MessageType := 'request-with-error message';
28.                perform SEND-POST-PROCESSING(MessageType,
                                              ParsedRequestsPtr);
29.          end if

30.       'a backend':
             /* There is a message from a backend for REQUEST-    */
             /* PREPARATION when                                  */
             /*    - a retrieve request caused by an update       */
             /*      request is completed                         */
             /*    - a record has changed cluster when executing  */
```

```
            /*     an update request.                          */
            /* get the message sent by the backend */
31.         perform RP-RECEIVE$BE-MESSAGE(MessagePtr);
            /* Generate a request. (This request is either an */
            /* insert or an update)                          */
32.         MessageType := ´backend message´;
33.         perform REQUEST-COMPOSER(MessageType,
                              MessagePtr,FormattedRequestsPtr);
            /* send the request to all the backends */
34.         perform BROADCAST-ALL-DIRECTORY-MANAGEMENT(
                                        FormattedRequestsPtr);

35.       ´otherwise´:
36.         system error;
37.    end case
38. end while


13.1 proc PARSER( input: TrafficUnitPtr,
               output: AllCorrect,ParsedRequestsPtr,
                 AggregateOperatorsMessagePtr,RequestCountMessagePtr);
        /* This routine parses all the requests in a traffic unit. (A */
        /* traffic unit is either a request or a transaction.)       */
        /*                                                           */
        /* TrafficUnitPtr :                                          */
        /*    a pointer to (traffic-id,TrafficUnit)                  */
        /* AllCorrect :                                              */
        /*    indicates whether or not all the requests in the traffic*/
        /*    unit are syntactically correct                         */
        /* ParsedRequestsPtr :                                       */
        /*    is one of the following two                            */
        /*       (1) a pointer to ((request-id,routing-indicator,    */
        /*           no-pred,parsed-request)[,...])  if all the      */
        /*           requests in the traffic unit are syntactically  */
        /*           correct.                                        */
        /*       (2) a pointer to ((request-id,request,              */
        /*           error-message)[,...])  if one (or more) of the  */
        /*           requests in the traffic unit is not syntactically*/
        /*           correct.                                        */
        /* AggregateOperatorsMessagePtr :                            */
        /*    a pointer to ((request-id,(attribute,                  */
        /*    aggregate-operator)[,...])[,...])                      */
        /* RequestCountMessagePtr :                                  */
        /*    a pointer to (traffic-id,request-count)                */
        /* request-count :                                           */
        /*    number of requests in a traffic nuit                   */
13.2 end proc


24.1 proc REQUEST-COMPOSER( input: MessageType,MessagePtr,
                          output: FormattedRequestsPtr);
        /* This routine transforms the requests into the form required*/
        /* for processing at the backends.                            */
        /*                                                            */
        /* MessageType :                                              */
        /*    indicates the type of a message                         */
        /* MessagePtr :                                               */
        /*    a pointer to a message. (It is either ParsedRequestsPtr */
        /*    or a pointer to a message sent by a backend.)           */
        /* ParsedRequestsPtr :                                        */
        /*    a pointer to ((request-id,routing-indicator,no-pred,    */
        /*    parsed-request)[,...])                                  */
        /* FormattedRequestsPtr :                                     */
        /*    a pointer to ((request-id,routing-indicator,            */
        /*    no-pred,sched-no,formatted-request)[,...])              */
24.2 end proc


module RP-RECEIVE
   programs  MESSAGE, SENDER, TRAFFIC-UNIT, BE-MESSAGE
   datasets  MessageBuffer
                   /* used to store messages for REQUEST-PREPARATION */

8.1 proc MESSAGE( input: nothing, output: nothing );
       /* This routine gets the next message for REQUEST-PREPARATION */
       /* and stores it in MessageBuffer.                            */
8.2 end proc

9.1 proc SENDER( input: nothing, output: sender);
       /* This routine returns the sender name of the next message   */
       /* for REQUEST-PREPARATION                                    */
       /*                                                            */
       /* sender :                                                   */
       /*   the sender name of the next message for REQUEST-         */
       /*   PREPARATION                                              */
9.2 end proc

12.1 proc TRAFFIC-UNIT( input: nothing, output: TrafficUnitPtr);
        /* This routine returns a pointer to the next traffic unit for*/
        /* REQUEST-PREPARATION                                        */
        /*                                                            */
        /* TrafficUnitPtr :                                           */
        /*    a pointer to (traffic-id,TrafficUnit)                   */
12.2 end proc

31.1 proc BE-MESSAGE( input: nothing, output: MessagePtr);
        /* This routine returnes a pointer to the next message for    */
        /* REQUEST-PREPARATION sent by a backend. There is a message  */
        /* from a backend for REQUEST-PREPARATION when                */
        /*    - a retrieve request caused by an update request is     */
        /*      completed                                             */
        /*    - a record has changed cluster when executing an update */
        /*      request                                               */
        /*                                                            */
        /* MessagePtr :                                               */
        /*    a pointer to a message                                     */
31.2 end proc

end module


17.1 proc SEND-POST-PROCESSING( input: MessageType,MessagePtr,
                               output: nothing);
        /* This routine sends a message to POST-PROCESSING               */
        /*                                                               */
        /* MessageType :                                                 */
        /*    indicates the type of a message                            */
        /* MessagePtr :                                                  */
        /*    a pointer to a message                                     */
17.2 end proc


25.1 proc BROADCAST-ALL-DIRECTORY-MANAGEMENT( input: FormattedRequestsPtr,
                                             output: nothing);
        /* This routine broadcasts a set of formatted requests to all */
        /* the backends                                                  */
        /* FormattedRequestsPtr :                                        */
        /*    a pointer to ((request-id,routing-indicator,no-pred,       */
        /*    sched-no,formatted-request)[,...])                         */
25.2 end proc


39. end task
```

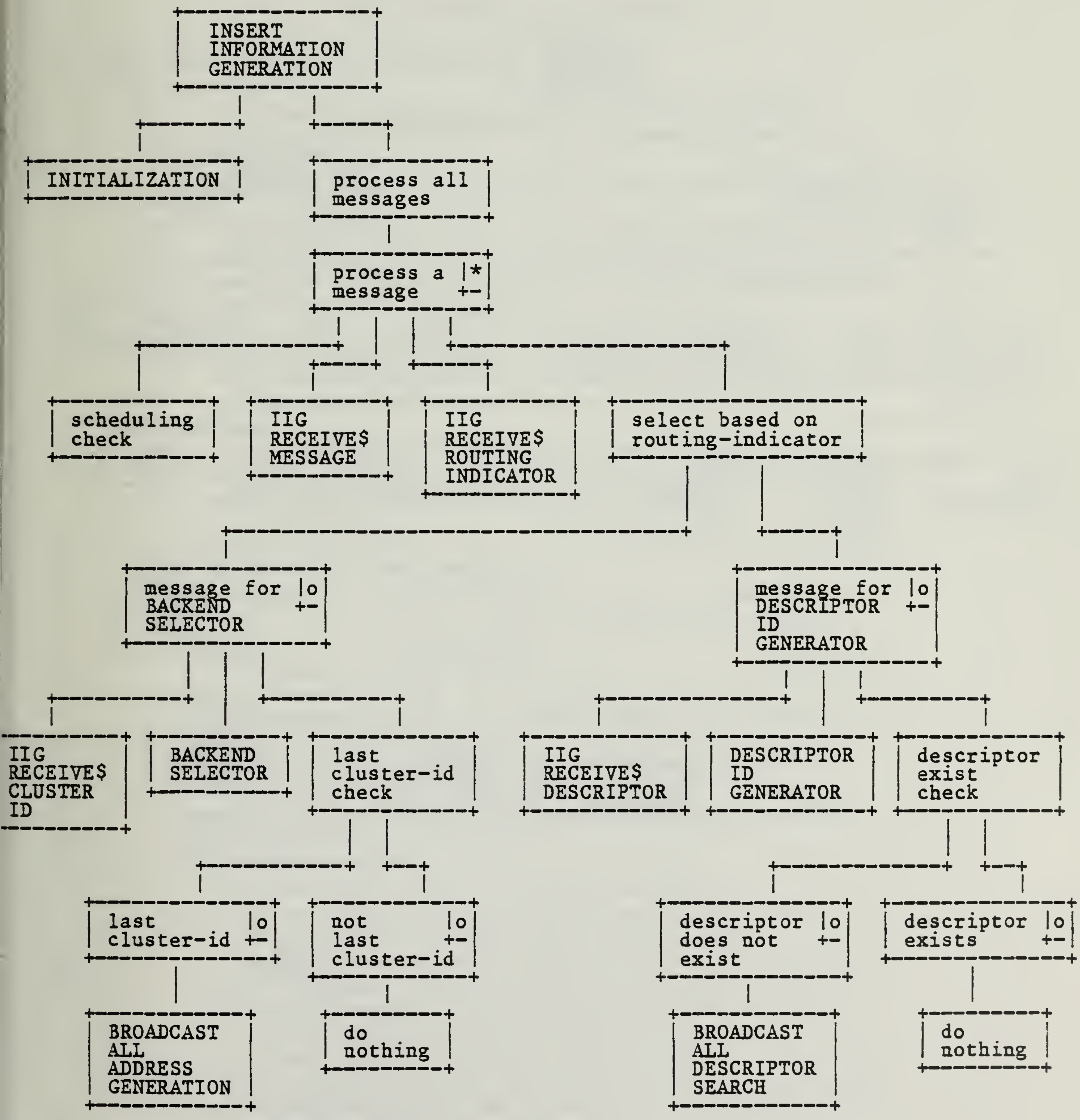
B.2 Part II - The Insert Information Generation Process

```
/*   (1) Part II    : The Insert Information Generation Process     */
/*   (2) Design     : INSERT-INFORMATION-GENERATION                 */
/*   (3) Designers  : A. Orooji,  Z.Z. Shi,  P.R. Strawser          */
/*   (4) Date       : Feb. 4, 1982                                  */
/*   (5) Modified   : Feb. 19, 1982                                 */
/*                    Apr. 12, 1982                                 */
/*                    Apr. 26, 1982                                 */
/*                    May  25, 1982                                 */
/*                    May  28, 1982                                 */
/*                    Jun.  2, 1982                                 */
/*   (6) Purpose    :                                               */
/*       This is the Insert Information Generation Process.  It     */
/* consists of the functions which must be performed during the     */
/* processing of an insert request to furnish additional            */
/* information required by the backends.                            */
```

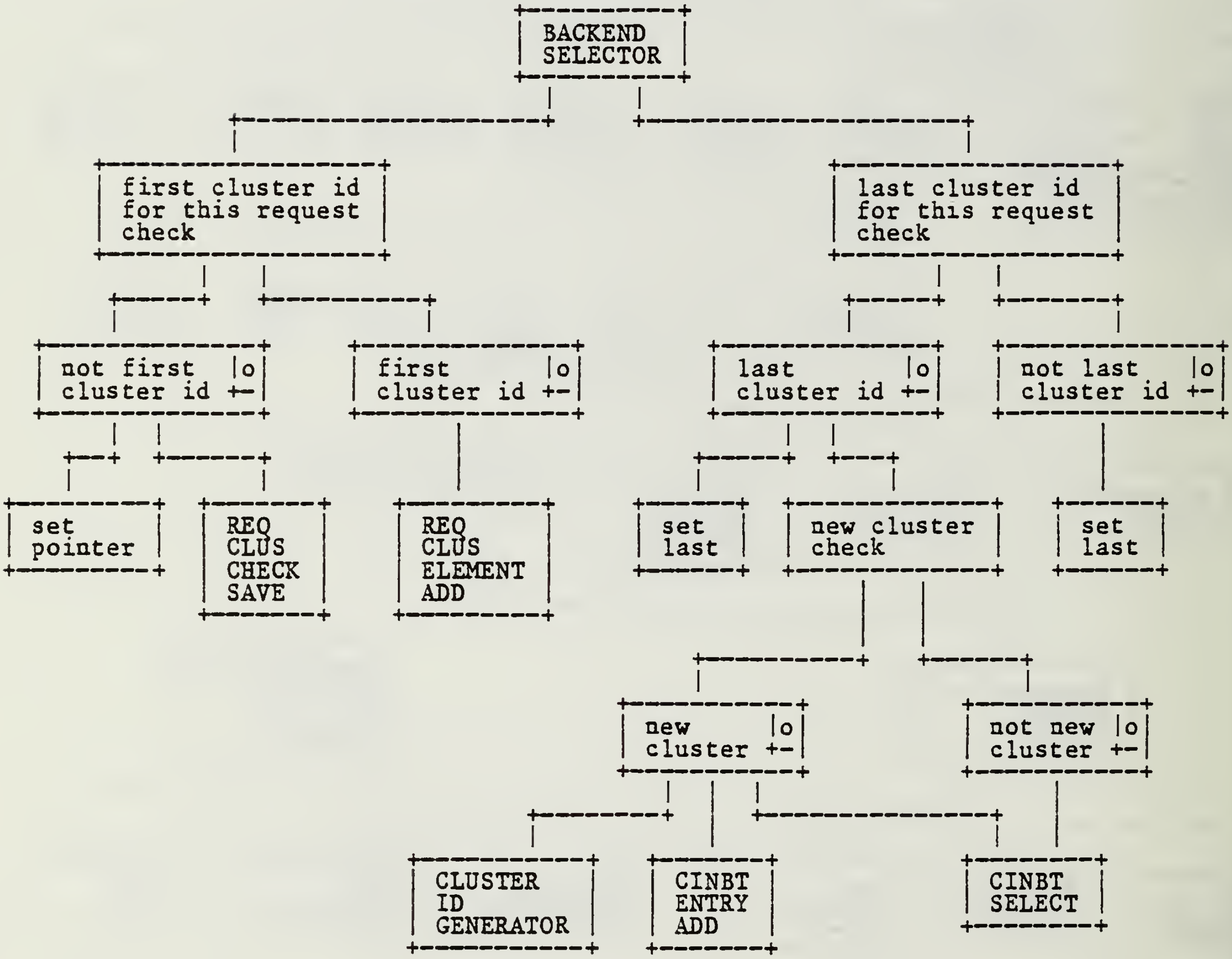
(8) Procedure Hierarchy for INSERT-INFORMATION-GENERATION

```
                      INSERT-INFORMATION-GENERATION
                                   |
    +-----------+--------+---------+----------+---------+------------+-----------+
    |           |        |         |          |         |            |           |
DESCRIPTOR  BACKEND   IIG       IIG        IIG        IIG         BROADCAST   BROADCA
ID          SELECTOR  RECEIVE$  RECEIVE$   RECEIVE$   RECEIVE$    ALL         ALL
GENERATOR       |     MESSAGE   ROUTING    CLUSTER    DESCRIPTOR  ADDRESS     DESCRIP
                |               INDICATOR  ID                     GENERATION  SEARCH
                |
     +----------+-------+------+--------+
     |          |       |      |        |
   CLUSTER    CINBT  CINBT   REQ      REQ
   ID         ENTRY  SELECT  CLUS     CLUS
   GENERATOR  ADD            ELEMENT  CHECK
                             ADD      SAVE
```

(9) Program Structure of INSERT-INFORMATION-GENERATION (Jackson Chart)



Program Structure of BACKEND-SELECTOR (Jackson Chart)



(10) Data Structures

```
/*   The data structure definitions are included in the program */
/* specifications.                                               */
```

(11) Program Specifications

```
1.  task INSERT-INFORMATION-GENERATION
       /* request-id :                                                */
       /*    consists of (traffic-id,request-no) which uniquely       */
       /*    identifies each request being processed by MDBS          */
       /* routing-indicator :                                         */
       /*    indicates where results/messages from backends should go to*/
       /* last :                                                      */
       /*    a flag ('true' or 'false') used when processing an insert */
       /*    request. This value is returned by BACKEND-SELECTOR and it */
       /*    indicates whether or not all the backends have returned a  */
       /*    cluster id (or a null value).                             */
       /* exist :                                                     */
       /*    a flag ('true' or 'false') used when processing a request */
       /*    for a new descriptor id. This value indicates whether or  */
       /*    not the descriptor already exists.                        */
       /* NewTrack :                                                  */
       /*    a flag ('true' or 'false') used when inserting a record.  */
       /*    This value indicates whether or not this is the first     */
       /*    record being inserted in a track.          .              */

2.     do initialization work;
3.     while 'true' do     /* do forever */
4.        if according to the task scheduling this task should
                                         release the processor
5.           then
6.              release the processor and wait;
7.        end if
          /* get the next message for INSERT-INFORMATION-GENERATION */
8.        perform IIG-RECEIVE$MESSAGE;
          /* get the routing-indicator in the next message */
          /* for INSERT-INFORMATION-GENERATION            */
9.        perform IIG-RECEIVE$ROUTING-INDICATOR(routing-indicator);
10.       case routing-indicator value
11.          'BACKEND-SELECTOR':
                /* an insert request is being executed */
                /* receive the request-id and the cluster id (or a */
                /* null value) returned by a backend              */
12.             perform IIG-RECEIVE$CLUSTER-ID(request-id,cluster-id);
                /* Determine a backend for record insertion if all */
                /* the backends have returned a cluster id (or a   */
                /* null value). Otherwise, save the cluster id (or */
                /* null value) returned by the backend.            */
                /* (BACKEND-SELECTOR will call CLUSTER-ID-GENERATOR*/
                /* when there is a new cluster.)                   */
13.             perform BACKEND-SELECTOR(request-id, cluster-id,
                                      last, backend-no, NewTrack);
14.             if last
15.                then
                      /* All the backends have returned a cluster  */
                      /* id (or a null value).                     */
                      /* Send the backend number selected for      */
                      /* record insertion to all the backends      */
16.                   perform BROADCAST-ALL-ADDRESS-GENERATION(request-id,
                                      backend-no, cluster-id, NewTrack);
17.             end if

18.          'DESCRIPTOR-ID-GENERATOR':
                /* request for new type-C subdescriptor */
                /* receive the request-id and the descriptor */
19.             perform IIG-RECEIVE$DESCRIPTOR(request-id,descriptor);
                /* The request for new type-C subdescriptor might   */
                /* have already been received  from another backend.*/
                /* In that case, the descriptor id has already been */
                /* generated and broadcasted.                       */
                /* Generate a descriptor id if it is not already    */
                /* generated                                        */
20.             perform DESCRIPTOR-ID-GENERATOR(descriptor,
                                           exist, descriptor-id);
21.             if exist = 'false'
22.                then
                      /* broadcast the descriptor id to all the backends */
23.                   perform BROADCAST-ALL-DESCRIPTOR-SEARCH(request-id,
                                                descriptor, descriptor-id);
24.             end if

25.          'otherwise':
26.             system error;
27.       end case
28.    end while


13.1  proc BACKEND-SELECTOR( input: request-id, input/output: cluster-id,
                            output: last,backend-no,NewTrack );
         /* This routine is called whenever a backend returns a cluster */
         /* id (or a null value). It determines a backend for record    */
         /* insertion when all the backends have returned a cluster id  */
         /* (or a null value). Otherwise, it saves the cluster id (or   */
         /* null value) returned by a backend.                          */
         /* (This routine will call CLUSTER-ID-GENERATOR when there is  */
         /* a new cluster.)                                             */
         /*                                                             */
         /* request-id :                                                */
         /*    consists of (traffic-id,request-no) which uniquely       */
         /*    identifies each request being processed by MDBS         */
         /* cluster-id :                                                */
         /*    uniquely identifies each cluster                         */
         /* NoBackends :                                                */
         /*    total number of backends in MDBS. (This is a variable    */
        /*    defined in SYSGEN.)                                     */
        /* last :                                                     */
        /*    a flag ('true' or 'false'). This value indicates whether */
        /*    or not all the backends have returned a cluster id (or a */
        /*    null value).                                            */
        /* backend-no :                                               */
        /*    the number of the backend minicomputer selected to insert*/
        /*    a new record into the database store                    */
        /* NewTrack :                                                 */
        /*    a flag ('true' or 'false'). This value indicates whether */
        /*    or not this is the first record being inserted in a track*/

        /*
         *  struct  request-cluster-id  {
         *              rid   /* (traffic-id,request-no) */
         *              cids-received-count  /* an integer indicating number
         *                     of cluster ids received for request rid */
         *              cid  /* cluster id received for request rid */
         *                   /* each cluster id received for a request is */
         *                   /* either null or has the same value as the  */
         *                   /* other non-null cluster ids received for   */
         *                   /* the request                               */
         *          }
         */

13.2    list  all-req-clus-id;  /* every element of this list consists
                         of the three parts in request-cluster-id */

13.3    if request-id = rid of one of the elements in all-req-clus-id list
13.4       then
              /* this is not the first cluster-id received for request */
              /* request-id                                            */
13.5          ptr := pointer to the element of all-req-clus-id
                                          with rid = request-id;
              /* save the cluster id received from the backend if needed */
13.6          perform REQ-CLUS-CHECK-SAVE(cluster-id, all-req-clus-id, ptr);
13.7       else
              /* this is the first cluster-id received for request  */
              /* request-id                                         */
              /* add an element to all-req-clus-id list             */
13.8          perform REQ-CLUS-ELEMENT-ADD(request-id,
                         cluster-id, all-req-clus-id, ptr);
13.9    end if
        /* check to see whether or not all the backends have */
        /* returned a cluster id (or a null value)          */
13.10   if cids-received-count < NoBackends
13.11      then
              /* not all the backends have returned a cluster */
              /* id (or a null value)                         */
13.12         last := 'false';
13.13      else
              /* all the backends have returned a cluster id (or */
              /* a null value)                                    */
```

```
13.14            last := ´true´;
13.15            if cid(ptr) = null
13.16               then
                       /* there is a new cluster */
13.17                  perform CLUSTER-ID-GENERATOR(cluster-id);
                       /* add an entry to cluster-id-to-next-backend table */
13.18                  perform CINBT-ENTRY-ADD(cluster-id);
13.19            end if
                 /* select a backend for record insertion */
13.20            perform CINBT-SELECT(cluster-id, backend-no, NewTrack);
13.21            delete element(ptr) in all-req-clus-id list;
13.22    end if
13.23 end proc
```

```
13.8.1 proc REQ-CLUS-ELEMENT-ADD( input: request-id,cluster-id,
                            input/output: all-req-clus-id, output: ptr );
          /* This routine adds an element to all-req-clus-id list */
13.8.2    list all-req-clus-id;
13.8.3    add an element to all-req-clus-id list;
13.8.4    ptr := pointer to the element just added;
13.8.5    ·rid(ptr) := request-id;
13.8.6    cid(ptr) := cluster-id;
13.8.7    cids-received-count(ptr) := 1;
13.8.8 end proc
```

```
13.6.1  proc REQ-CLUS-CHECK-SAVE( input: cluster-id,ptr,
                            input/output: all-req-clus-id, ouput: nothing);
           /* This routine saves the cluster id received from a */
           /* backend if needed                                 */
13.6.2     list all-req-clus-id;
13.6.3     cids-received-count(ptr) := cids-received-count(ptr) + 1;
13.6.4     if cluster-id ~= null
13.6.5        then
                 /* cluster id received from the backend is not null */
13.6.6           if cid(ptr) = null
13.6.7              then
                       /* all the previous cluster ids received from */
                       /* backends are null                          */
                       /* cluster-id ~= null,  cid(ptr) = null  */
13.6.8                 cid(ptr) := cluster-id;
13.6.9              else
                       /* cluster-id ~= null,  cid(ptr) ~= null   */
13.6.10                if cluster-id ~= cid(ptr)
13.6.11                   then
                             /* cluster-id ~= null, cid(ptr) ~= null, */
                             /* cluster-id ~= cid(ptr)                */
13.6.12                      ´system error´;
13.6.13                end if
13.6.14          end if
13.6.15    end if
13.6.16 end proc
```

```
13.18.1 proc CINBT-ENTRY-ADD( input: cluster-id, output: nothing );
            /* This routine adds an entry to cluster-id-to-next-backend */
            /* table                                                    */
13.18.2     add an entry to cluster-id-to-next-backend table;
                    /* cluster id for this entry is cluster-id   */
                    /* backend number for this entry is a random */
                    /* number bn ( 1 =< bn =< NoBackends )       */
13.18.3 end proc


13.20.1 proc CINBT-SELECT( input: cluster-id, output: backend-no,NewTrack);
            /* This routine selects a backend for record insertion */
13.20.2     backend-no := backend selected (using CINBT) for record insertion;
13.20.3     set NewTrack (using CINBT);
13.20.4     update CINBT if needed;
13.20.5 end proc


13.17.1 proc CLUSTER-ID-GENERATOR( input: nothing, output: cluster-id );
            /* This routine generates a new cluster id.             */
            /*                                                      */
            /* cluster-id :                                         */
            /*     uniquely identifies each cluster                 */
13.17.2 end proc


20.1  proc DESCRIPTOR-ID-GENERATOR( input: descriptor,
                                    output: exist,descriptor-id );
        /* This routine generates a new descriptor id for a descriptor */
        /* if it is not already generated.                             */
        /*                                                             */
        /* exist :                                                     */
        /*    a flag ('true' or 'false'). This value indicates whether */
        /*    or not the descriptor already exists.                    */
        /* descriptor-id :                                             */
        /*    uniquely identifies each descriptor                      */

        /* This first implementation of DESCRIPTOR-ID-GENERATOR keeps */
        /* the descriptor ids generated from system start up to system*/
        /* shut down (or until database reorganization). We need to   */
        /* keep the descriptor ids generated because multiple backends*/
        /* may request a descriptor id for the same descriptor.       */

        /*
         * struct descriptor-descriptor-id {
         *           desc  /* descriptor */
         *           desc-id  /* corresponding descriptor id */
         *       }
         */

20.2    list  desc-desc-id-list;  /* every element of this list
```

```
                     has the two elements in descriptor-descriptor-id */
20.3     if descriptor = desc of one of the elements in desc-desc-id-list
20.4        then
               /* the descriptor id is already generated */
20.5           exist := 'true';
20.6        else
               /* the descriptor id is not generated yet */
20.7           exist := 'false';
20.8           descriptor-id := generate a descriptor id;
20.9           add (descriptor,descriptor-id) to desc-desc-id-list;
20.10    end if
20.11 end proc


module IIG-RECEIVE
   programs  MESSAGE, ROUTING-INDICATOR, CLUSTER-ID, DESCRIPTOR
   datasets  MessageBuffer
               /* used to save messages for INSERT-INFORMATION-GENERATION */

8.1 proc MESSAGE( input: nothing, output: nothing );
       /* This routine gets the next message for INSERT-INFORMATION-    */
       /* GENERATION and stores it in MessageBuffer.                     */
8.2 end proc

9.1 proc ROUTING-INDICATOR( input: nothing,
                                    output: routing-indicator);
       /* This routine returns the routing-indicator in the next         */
       /* message for INSERT-INFORMATION-GENERATION.                     */
       /*                                                                */
       /* routing-indicator :                                            */
       /*    indicates where results/messages from backends should go to*/
9.2 end proc

12.1 proc CLUSTER-ID( input: nothing, output: request-id,cluster-id);
        /* This routine returns the request-id and the cluster id (or a */
        /* null value) returned by a backend.                            */
        /*                                                               */
        /* request-id :                                                  */
        /*    consists of (traffic-id,request-no) which uniquely         */
        /*    identifies each request being processed by MDBS            */
        /* cluster-id :                                                  */
        /*    uniquely identifies each cluster                           */
12.2 end proc

19.1 proc DESCRIPTOR( input: nothing, output: request-id,descriptor);
        /* This routine returns the descriptor sent by a backend.        */
        /*                                                               */
        /* request-id :                                                  */
        /*    consists of (traffic-id,request-no) which uniquely         */
        /*    identifies each request being processed by MDBS            */
19.2 end proc

end module
```

```
16.1 proc BROADCAST-ALL-ADDRESS-GENERATION( input: request-id,
                              backend-no,cluster-id, output: nothing);
        /* This routine broadcasts the backend number selected for record*/
        /* insertion to all the backends.                                 */
        /*                                                                */
        /* request-id :                                                   */
        /*    consists of (traffic-id,request-no) which uniquely          */
        /*    identifies each request being processed by MDBS             */
        /* backend-no :                                                   */
        /*    the number of the backend minicomputer selected to insert   */
        /*    a new record into the database store                        */
        /* cluster-id :                                                   */
        /*    uniquely identifies each cluster                            */
16.2 end proc
```

```
23.1 proc BROADCAST-ALL-DESCRIPTOR-SEARCH( input: request-id,
                               descriptor,descriptor-id, output: nothing);
        /* This routine broadcasts a new descriptor id to all the backends.*/
        /*                                                                  */
        /* request-id :                                                     */
        /*    consists of (traffic-id,request-no) which uniquely identifies*/
        /*    each request being processed by MDBS                          */
        /* descriptor-id :                                                  */
        /*    uniquely identifies each descriptor                           */
23.2 end proc


29. end task
```

B.3 Part III - The Post Processing Process

```
/*   (1) Part III   : The Post Processing Process                 */
/*   (2) Design     : POST-PROCESSING                             */
/*   (3) Designers : A. Orooji,  Z.Z. Shi,  P.R. Strawser         */
/*   (4) Date       : Feb. 4, 1982                                */
/*   (5) Modified   : Feb. 19, 1982                               */
/*                    Apr. 12, 1982                               */
/*                    Apr. 26, 1982                               */
/*                    Jun. 22, 1982                               */
/*                    Jun. 25, 1982                               */
/*                    Jul.  8, 1982                               */
/*   (6) Purpose    :                                             */
/*      This is the Post Processing process.  It consists of the  */
/* functions which must be performed before the results of a      */
/* request or a transaction are forwarded to the host machine.    */
```

```
                   Procedure Hierarchy for POST-PROCESSING
                                  POST-PROCESSING
                                        |
  +-------------------+-----------------+-------------------------+
  |                   |                 |                         |
 PP                  PP                PP                         PP
 RECEIVE$            RECEIVE$          CTRL                       BE
 MESSAGE             SENDER            TASK                       TASK
                                       MSG                        MSG
                                        |                          |
 +---------+---------+--------+---------+-----+----------+         |
 |         |         |        |         |     |          |         |
PP        PP        PP       PP        PP    PP        REPLY       |
RECEIVE$  RECEIVE$  REQUEST  RECEIVE$  AGGR  RECEIVE$  MONITOR     |
MESSAGE   REQ       COUNT$   AGGR      OP$   ERROR                 |
TYPE      COUNT     SAVE     OP        SAVE  MSG                   |
                                                                   |
         +----------+----------+-------+-----------+---------------+
         |          |          |       |           |
        PP         PP        REPLY    PP        AGGREGATE
        RECEIVE$   RECEIVE$  MONITOR  RECEIVE$  POST
        ROUTING    RESULTS            PARTIAL   OPERATION
        INDICATOR                     RESULTS


                         REPLY-MONITOR
                                |
                      +---------+---------------+
                      |                         |
            SEND-COMPLETION-SIGNAL       PP-RESULTS$STORE
                      |
        PP-REQUEST-COUNT$LAST-REQUEST-CHECK
```

Program Structure of POST-PROCESSING (Jackson Chart)

```
             +------------+
             | POST       |
             | PROCESSING |
             +------------+
                 |   |
        +--------+   +------+
        |                   |
+----------------+    +--------------+
| INITIALIZATION |    | process all  |
+----------------+    | messages     |
                      +--------------+
                             |
                      +--------------+
                      | process a  |*|
                      | message    +-|
                      +--------------+
                        |  |  |  |
      +-----------------+  |  |  +-------------------+
      |               +----+  +----+                 |
      |               |            |                 |
+------------+  +----------+  +----------+  +------------------+
| scheduling |  | PP       |  | PP       |  | select based on  |
| check      |  | RECEIVE$ |  | RECEIVE$ |  | message sender   |
+------------+  | MESSAGE  |  | SENDER   |  +------------------+
                +----------+  +----------+          |   |
                                                    |   |
                                                    |   |
                              +---------------------+   +---------+
                              |                                   |
                       +--------------+                   +--------------+
                       | a task in  |o|                   | a task in  |o|
                       | controller +-|                   | a backend  +-|
                       +--------------+                   +--------------+
                              |                                   |
                          +--------+                          +--------+
                          |  PP    |                          |  PP    |
                          |  CTRL  |                          |  BE    |
                          |  TASK  |                          |  TASK  |
                          |  MSG   |                          |  MSG   |
                          +--------+                          +--------+
```

Program Structure of PP-CTRL-TASK-MSG (Jackson Chart)

```
                   +----------+
                   |   PP     |
                   |   CTRL   |
                   |   TASK   |
                   |   MSG    |
                   +----------+
                       |  |
         +-------------+  +----------+
         |                           |
    +----------+               +----------+
    | PP       |               | select   |
    | RECEIVE$ |               | based on |
    | MESSAGE  |               | message  |
    | TYPE     |               | type     |
    +----------+               +----------+
                                   |  |  |
          +------------------------+  |  +----------------------+
          |                           |                         |
  +------------------+        +--------------+           +-------------+
  | number of      |o|        | aggregate  |o|           | requests  |o|
  | requests in a  +-|        | operators  +-|           | with      +-|
  | traffic unit     |        +--------------+           | errors      |
  +------------------+                                   +-------------+
         |   |                     |   |                       |   |
     +---+   +---+              +--+   +-----+             +---+   +---+
     |           |              |            |             |           |
+-----------++----------+   +-----------++-------+   +-----------++----------+
| PP        || PP       |   | PP        || PP    |   | PP        || REPLY    |
| RECEIVE$  || REQUEST  |   | RECEIVE$  || AGGR  |   | RECEIVE$  || MONITOR  |
| REQ       || COUNT$   |   | AGGR      || OP$   |   | ERROR     |+----------+
| COUNT     || SAVE     |   | OP        || SAVE  |   | MSG       |
+-----------++----------+   +-----------++-------+   +-----------+
```

Program Structure of PP-BE-TASK-MSG (Jackson Chart)

```
               +-----------+
               |   PP      |
               |   BE      |
               |   TASK    |
               |   MSG     |
               +-----------+
                    |   |
            +------+   +-----------+
            |                      |
      +------------+     +--------------------+
      | PP         |     | select             |
      | RECEIVE$   |     | based on           |
      | ROUTING    |     | routing-indicator  |
      | INDICATOR  |     +--------------------+
      +------------+             |    |
                                 |    |
             +-------------------+    +-----------+
             |                                    |
       +-----------+                      +---------------+
       | message |o|                      | message for |o|
       | for     +-|                      | AGGREGATE   +-|
       | REPLY     |                      | POST          |
       | MONITOR   |                      | OPERATION     |
       +-----------+                      +---------------+
           |  |                                |   |  |
     +-----+  +-----+               +----------+   |  +--------+
     |              |               |              |           |
+----------+   +----------+    +-----------+ +-----------+ +---------+
| PP       |   | REPLY    |    | PP        | | AGGREGATE | | results |
| RECEIVE$ |   | MONITOR  |    | RECEIVE$  | | POST      | | ready   |
| RESULTS  |   +----------+    | PARTIAL   | | OPERATION | | check   |
+----------                    | RESULTS   | +-----------+ +---------+
                               +-----------+                  |  |
                                                              |  |
                                                        +-----+  +----+
                                                        |             |
                                                   +-----------+ +-----------+
                                                   | results |o| | results |o|
                                                   | ready   +-| | not     +-|
                                                   +-----------+ | ready     |
                                                         |       +-----------+
                                                         |             |
                                                    +---------+   +---------+
                                                    | REPLY   |   | do      |
                                                    | MONITOR |   | nothing |
                                                    +---------+   +---------+
                   Program Structure of REPLY-MONITOR (Jackson Chart)

                               +----------------+
                               | REPLY-MONITOR  |
                               +----------------+
                                       |
                             +-------------------+
                             | select            |
                             | based on          |
                             | routing-indicator |
                             +-------------------+
                                   |  |  |
           +-----------------------+  |  +-----------------------------+
           |                          |                                |
  +------------------+      +------------------+             +------------------+
  | requests-with- |o|      | AGGREGATE-POST |o|             | REPLY-MONITOR  |o|
  | error          +-|      | OPERATION      +-|             | message        +-|
  | message          |      | message          |             +------------------+
  +------------------+      +------------------+                    |  |  |
        |                        |   |                              |  |  |
        |                  +-----+   +----+               +---------+  |  +--------+
        |                  |              |               |            |           |
+--------------+  +-------------+ +------------+   +-----------++-----------+ +---------+
| send error   |  | send the    | | SEND       |   | PP        || Results-  | | last    |
| messages to  |  | results to  | | COMPLETION |   | RESULTS$  || Buffer    | | results |
| the host     |  | the host    | | SIGNAL     |   | STORE     || full      | | check   |
| machine      |  | machine     | +------------+   +-----------+| check     | +---------+
+--------------+  +-------------+                               +-----------+    |   |
                                                                     |   |       |   |
              +------------------------------------------------------+   |       |   |
              |                      +-----------------------------------+       |   |
              |                      |                        +------------------+   |
              |                      |                        |                   +--+
              |                      |                        |                   |.
     +-----------------+   +-----------------+          +-----------+     +------------+
     | ResultsBuffer |o|   | ResultsBuffer |o|          | last    |o|     | not last |o|
     | full          +-|   | not full      +-|          | results +-|     | results  +-|
     +-----------------+   +-----------------+          +-----------+     +------------+
              |                     |                       |  |                   |
              |                     |               +-------+  +----+              |
              |                     |               |               |              |
        +-------------+      +----------+     +-------------++------------+  +----------+
        | send the    |      | do       |     | send the    || SEND       |  | do       |
        | results to  |      | nothing  |     | results to  || COMPLETION |  | nothing  |
        | the host    |      +----------+     | the host    || SIGNAL     |  +----------+
        | machine     |                       | machine     |+------------+
        +-------------+                       +-------------+
```

(10) Data Structures

```
/*   The data structure definitions are included in the program */
/* specifications.                                              */
```

(11) Program Specifications

```
1.  Task POST-PROCESSING
       /* sender :                                                     */
       /*    the sender name of the next message for POST-PROCESSING;   */
       /*    the possible values are: ´a task in the controller´ and ´a */
       /*    task in a backend´                                         */

2.     do initialization work;
3.     while ´true´ do     /* do forever */
4.        if according to the task scheduling this task should release
                                                          the processor
5.           then
6.              release the processor and wait;
7.        end if
          /* get the next message for POST-PROCESSING */
8.        perform PP-RECEIVE$MESSAGE;
          /* get the sender name of the next message for POST-PROCESSING */
9.        perform PP-RECEIVE$SENDER(sender);
10.       case sender value
11.          ´a task in the controller´:
12.             perform PP-CTRL-TASK-MSG;
13.          ´a task in a backend´:
14.             perform PP-BE-TASK-MSG;
15.          ´otherwise´:
16.             system error;
17.       end case
18.    end while


12.1  proc PP-CTRL-TASK-MSG ( input: nothing, output: nothing );
        /* This routine is used when there is a message for POST-          */
        /* PROCESSING from a task in the controller.                        */
        /*                                                                  */
        /* TrafficUnit :                                                    */
        /*    is either a request or a transaction                          */
        /* traffic-id :                                                     */
        /*    an integer that identifies a traffic unit                     */
        /* request-id :                                                     */
        /*    consists of (traffic-id,request-no) which uniquely            */
        /*    identifies each request being processed by MDBS               */
        /* MessageType :                                                    */
        /*    indicates the type of a message (from a task in the           */
        /*    controller) for POST-PROCESSING; the possible values are:    */
        /*    ´number-of-requests-in-a-traffic-unit message´, ´aggregate-*/
        /*    operators message´ and ´requests-with-error message´         */
        /* RequestCountMessagePtr :                                         */
```

```
         /*    a pointer to (traffic-id,request-count)                     */
         /* request-count :                                                */
         /*    number of requests in a traffic unit                        */
         /* AggregateOperatorsMessagePtr :                                 */
         /*    a pointer to ((request-id,(attribute,                       */
         /*          aggregate-operator)[,...])[,...])                     */
         /* RequestsWithErrorPtr :                                         */
         /*    a pointer to ((request-id,request,error-message)[,...])     */
         /* routing-indicator :                                            */
         /*    indicates the type of the results sent to REPLY-MONITOR;    */
         /*    its value is: ´requests-with-error message´                 */

         /* The message is from REQUEST-PREPARATION.                       */
         /* There is a MessageType that indicates the message contains     */
         /*    - number of requests in a traffic unit (provided by PARSER)*/
         /*    - error messages (provided by PARSER)                       */
         /*    - aggregate operators in a retrieve request (provided by    */
         /*      PARSER)                                                   */
12.2     perform PP-RECEIVE$MESSAGE-TYPE(MessageType);
12.3     case MessageType value
12.4        ´number-of-requests-in-a-traffic-unit message´:
12.5           perform PP-RECEIVE$REQ-COUNT(RequestCountMessagePtr);
               /* save the information to be used by REPLY-MONITOR later */
12.6           perform PP-REQUEST-COUNT$SAVE(RequestCountMessagePtr);

12.7        ´aggregate-operators message´:
12.8           perform PP-RECEIVE$AGGR-OP(AggregateOperatorsMessagePtr);
               /* save the information to be used by AGGREGATE-POST- */
               /* OPERATION later                                    */
12.9           perform PP-AGGR-OP$SAVE(AggregateOperatorsMessagePtr);

12.10       ´requests-with-error message´:
12.11          perform PP-RECEIVE$ERROR-MSG(RequestsWithErrorPtr);
               /* send the error messages to the host machine */
12.12          routing-indicator := ´requests-with-error message´;
12.13          perform REPLY-MONITOR(routing-indicator, NULL,
                                            RequestsWithErrorPtr);

12.14       ´otherwise´:
12.15          system error;
12.16    end case
12.17 end proc


14.1  proc PP-BE-TASK-MSG ( input: nothing, output: nothing );
         /* This routine is used when there is a message for POST-     */
         /* PROCESSING from a task in a backend.                       */
         /*                                                            */
         /* request-id :                                               */
         /*    consists of (traffic-id,request-no) which uniquely      */
         /*    identifies each request being processed by MDBS         */
         /* routing-indicator :                                        */
         /*    indicates where the results should go to; it also      */
         /*    indicates the type of the results; the possible values */
         /*    are: 'REPLY-MONITOR' and 'AGGREGATE-POST-OPERATION'    */
         /* Results :                                                 */
         /*    results returned from a backend                        */
         /* PartialResults :                                          */
         /*    partial results returned from a backend (for retrieve  */
         /*    requests with aggregate operators)                     */
         /* last :                                                    */
         /*    a flag ('true' or 'false'); it indicates whether or not*/
         /*    all the backends have returned their results           */
         /* AggregateResultsPtr :                                     */
         /*    a pointer to the results computed by AGGREGATE-POST-   */
         /*    OPERATION from partial results                         */

         /* Get the request-id and the routing-indicator in the message */
14.2     perform PP-RECEIVE$ROUTING-INDICATOR(request-id, routing-indicator);
14.3     case routing-indicator value
14.4        'REPLY-MONITOR':
               /* receive the results returned by the backend */
14.5           perform PP-RECEIVE$RESULTS(Results);
               /* send the results to the host machine */
14.6           perform REPLY-MONITOR(routing-indicator, request-id, Results);

14.7        'AGGREGATE-POST-OPERATION':
               /* receive the partial results returned by the backend */
14.8           perform PP-RECEIVE$PARTIAL-RESULTS(PartialResults);
14.9           perform AGGREGATE-POST-OPERATION(request-id,
                        PartialResults, AggregateResultsPtr, last);
14.10          if last
14.11             then
                     /* The results are ready.              */
                     /* Send the results to the host machine. */
14.12                perform REPLY-MONITOR(routing-indicator, request-id,
                                                 AggregateResultsPtr);
14.13          end if

14.14       'otherwise':
               system error;
14.15
14.16    end case
14.17 end proc


12.13.1  proc REPLY-MONITOR(input: routing-indicator,request-id,Results,
                          output: nothing);
            /* This routine sends the results to the host machine. It    */
            /* will also send a completion signal to the host machine    */
            /* when all the results have been sent.                      */
            /*                                                           */
            /* routing-indicator :                                       */
            /*    indicates the type of results; the possible values are:*/
            /*    'requests-with-error message', 'REPLY-MONITOR' and     */
            /*    'AGGREGATE-POST-OPERATION'                             */
            /* request-id :                                              */
```

```
             /*    consists of (traffic-id,request-no) which uniquely     */
             /*    identifies each request being processed by MDBS        */
             /* Results :                                                 */
             /*    is one of the following two                            */
             /*       - the results returned from a backend               */
             /*       - a pointer to the results when the results are from*/
             /*         either PARSER or AGGREGATE-POST-OPERATION         */
             /* last :                                                    */
             /*    a flag ('true' or 'false'); it indicates whether or not*/
             /*    all the backends have returned their results           */
             /* BufferFull :                                              */
             /*    a flag ('true' or 'false'); it indicates whether or not*/
             /*    the buffer used for storing the results returned by the*/
             /*    backends is full                                       */

12.13.2      case routing-indicator value
12.13.3         'requests-with-error message':
12.13.4            send the error messages to the host machine;
                       /* Results is pointing to the results */

12.13.5         'AGGREGATE-POST-OPERATION':
12.13.6            send the results of aggregations to the host machine;
                       /* Results is pointing to the results */
                   /* send a completion signal to the host machine */
12.13.7            perform SEND-COMPLETION-SIGNAL(request-id);

12.13.8         'REPLY-MONITOR':
                   /* store the results returned by the backend in a buffer */
12.13.9            perform PP-RESULTS$STORE(request-id, routing-indicator,
                                            Results, BufferFull, last);
12.13.10           if BufferFull
12.13.11              then
                         /* the buffer used for storing the results is full */
12.13.12                 send the results to the host machine;
12.13.13           end if
12.13.14           if last
12.13.15              then
                         /* All the backends have returned their results. */
                         /* Send the results remaining in the buffer      */
                         /* to the host machine.                          */
12.13.16                 send the results to the host machine;
                         /* All the results for the request have been sent*/
                         /* to the host machine.                          */
                         /* Send a completion signal to the host machine. */
12.13.17                 perform SEND-COMPLETION-SIGNAL(request-id);
12.13.18           end if

12.13.19        'otherwise':
12.13.20           system error;
12.13.21     end case
12.13.22 end proc
```

```
12.13.7.1  proc SEND-COMPLETION-SIGNAL( input: request-id, output: nothing);
              /* This routine sends a request-completion signal to the   */
              /* host machine.  It also checks to see whether all the    */
              /* results for a transaction have been sent to the host    */
              /* machine. If so, it sends a transaction-completion signal*/
              /* to the host machine.                                    */
              /*                                                         */
              /* request-id :                                            */
              /*    consists of (traffic-id,request-no) which uniquely   */
              /*    identifies each request being processed by MDBS      */
              /* TransactionDone :                                       */
              /*    a flag ('true' or 'false'); it indicates whether or  */
              /*    not all the requests in a transaction have been      */
              /*    completed                                            */
              /* NonTransaction :                                        */
              /*    a constant that is assigned to request-no of a       */
              /*    request that is not part of a transaction.  (We      */
              /*    recall that a request is identified by its request-id*/
              /*    which is (traffic-id,request-no).  If the request is */
              /*    not part of a transaction, traffic-id identifies the */
              /*    request and request-no can be set to NonTransaction. */
              /*    By doing this, we will be able to tell whether or not*/
              /*    a request is part of a transaction.)                 */

12.13.7.2     send a request-completion signal to the host machine;
              /* Check to see if the request is part of a transaction */
12.13.7.3     if request-no ~= NonTransaction
12.13.7.4        then
                    /* The request is part of a transaction.         */
                    /* Indicate that one of the requests in the traffic */
                    /* unit has been completed and check to see whether */
                    /* all the results for the traffic unit have been   */
                    /* sent to the host machine.                        */
12.13.7.5           perform PP-REQUEST-COUNT$LAST-REQUEST-CHECK(traffic-id,
                                                          TransactionDone);
12.13.7.6           if TransactionDone
12.13.7.7              then
                          /* All the results for the traffic unit have */
                          /* been sent to the host machine.            */
12.13.7.8                 send a transaction-completion signal to the
                                                             host machine;
12.13.7.9           end if
12.13.7.10    end if
12.13.7.11 end proc


14.9.1 proc AGGREGATE-POST-OPERATION( input: request-id,PartialResults,
                                     output: AggregateResultsPtr,last );
          /* This routine performs the aggregate operations on the     */
          /* partial results returned by the backends.  It will set    */
          /* 'last' to indicate whether or not all the backends have   */
          /* returned their partial results.                           */
          /*                                                           */
          /* request-id :                                               */
          /*    consists of (traffic-id,request-no) which uniquely      */
          /*    identifies each request being processed by MDBS         */
          /* PartialResults :                                           */
          /*    partial results returned from a backend (for retrieve   */
          /*    requests with aggregate operators)                      */
          /* AggregateResultsPtr :                                      */
          /*    a pointer to the results computed by AGGREGATE-POST-    */
          /*    OPERATION from partial results                          */
          /* last :                                                     */
          /*    a flag ('true' or 'false'); it indicates whether or not*/
          /*    all the backends have returned their results            */
          /* NoBackends :                                               */
          /*    total number of backends in MDBS. (This is a variable   */
          /*    defined in SYSGEN.)                                     */
14.9.2 end proc



module PP-RECEIVE
   programs  MESSAGE, SENDER, MESSAGE-TYPE, REQ-COUNT, AGGR-OP,
                 ERROR-MSG, ROUTING-INDICATOR, RESULTS, PARTIAL-RESULTS
   datasets  MessageBuffer
                  /* used to store messages for POST-PROCESSING */

8.1 proc MESSAGE ( input: nothing, output: nothing );
       /* This routine gets the next message for POST-PROCESSING    */
       /* and stores it in MessageBuffer.                           */
8.2 end proc

9.1 proc SENDER ( input: nothing, output: sender );
       /* This routine returns the sender name of the next message for */
       /* POST-PROCESSING.                                              */
       /*                                                               */
       /* sender :                                                      */
       /*    the sender name of the next message for POST-PROCESSING;   */
       /*    the possible values are: 'a task in the controller' and    */
       /*    'a task in a backend'                                      */
9.2 end proc

12.2.1 proc MESSAGE-TYPE ( input: nothing, output: MessageType );
          /* This routine returns the message type of the next message     */
          /* (from a task in the controller) for POST-PROCESSING.          */
          /*                                                               */
          /* MessageType :                                                 */
          /*    indicates the type of a message (from a task in the        */
          /*    controller) for POST-PROCESSING; the possible values are:  */
          /*    'number-of-requests-in-a-traffic-unit message',            */
          /*    'aggregate-operators message' and 'requests-with-error     */
          /*    message'                                                   */
12.2.2 end proc

12.5.1 proc REQ-COUNT ( input: nothing, output: RequestCountMessagePtr );
```

```
          /* This routine returns RequestCountMessagePtr sent by REQUEST- */
          /* PREPARATION.                                                 */
          /*                                                              */
          /* RequestCountMessagePtr :                                     */
          /*    a pointer to (traffic-id,request-count)                   */
          /* request-count :                                              */
          /*    number of requests in a traffic unit                      */
12.5.2 end proc

12.8.1 proc AGGR-OP ( input: nothing, output: AggregateOperatorsMessagePtr);
          /* This routine returns AggregateOperatorsMessagePtr sent by    */
          /* REQUEST-PREPARATION.                                         */
          /*                                                              */
          /* AggregateOperatorsMessagePtr :                               */
          /*    a pointer to ((request-id,(attribute,                     */
          /*                        aggregate-operator)[,...])[,...])     */
12.8.2 end proc

12.11.1 proc ERROR-MSG ( input: nothing, output: RequestsWithErrorPtr);
           /* This routine returns RequestsWithErrorPtr sent by REQUEST-   */
           /* PREPARATION.                                                 */
           /*                                                              */
           /* RequestsWithErrorPtr :                                       */
           /*    a pointer to ((request-id,request,error-message)[,...])   */
12.11.2 end proc

14.2.1 proc ROUTING-INDICATOR ( input: nothing,
                               output: request-id,routing-indicator );
          /* This routine returns the request-id and the routing-indicator*/
          /* in the next message (from a backend) for POST-PROCESSING.    */
          /*                                                              */
          /* request-id :                                                 */
          /*    consists of (traffic-id,request-no) which uniquely        */
          /*    identifies each request being processed by MDBS           */
          /* routing-indicator :                                          */
          /*    indicates where the results should go to; it also         */
          /*    indicates the type of the results; the possible values    */
          /*    are: 'REPLY-MONITOR' and 'AGGREGATE-POST-OPERATION'       */
14.2.2 end proc

14.5.1 proc RESULTS ( input: nothing, output: Results );
          /* This routine returns the results sent by a backend.          */
          /*                                                              */
          /* Results :                                                    */
          /*    results returned from a backend                           */
14.5.2 end proc

14.8.1 proc PARTIAL-RESULTS ( input: nothing, output: PartialResults);
          /* This routine returns the partial results sent by a backend.  */
          /*                                                              */
          /* PartialResults :                                             */
          /*    partial results returned from a backend                   */
14.8.2 end proc
```

```
end module



module PP-REQUEST-COUNT
   programs  SAVE, LAST-REQUEST-CHECK
   datasets  CountBuffer   /* used to save RequestCountMessagePtr's  */

12.6.1 proc SAVE ( input: RequestCountMessagePtr, output: nothing );
         /* This routine saves RequestCountMessagePtr to be used by     */
         /* LAST-REQUEST-CHECK later.                                   */
         /*                                                             */
         /* RequestCountMessagePtr :                                    */
         /*    a pointer to (traffic-id,request-count)                  */
         /* request-count :                                             */
         /*    number of requests in a traffic unit                     */
12.6.2 end proc

12.13.7.5.1  proc LAST-REQUEST-CHECK( input: traffic-id,
                                      output: TransactionDone);
               /* This routine remembers that one of the requests in the */
               /* traffic unit has been completed.  It will set          */
               /* 'TransactionDone' to indicate whether or not all the   */
               /* requests in the traffic unit have been completed.      */
               /*                                                        */
               /* traffic-id : .                                         */
               /*    an integer that identifies a traffic unit           */
               /* TransactionDone :                                      */
               /*    a flag ('true' or 'false'); it indicates whether or */
               /*    not all the requests in a traffic unit have been    */
               /*    completed                                           */

12.13.7.5.2    remember that one of the requests in the traffic unit has
                                                         been completed;
12.13.7.5.3    if all the requests in the traffic unit have been completed
12.13.7.5.4       then
12.13.7.5.5          TransactionDone := 'true';
12.13.7.5.6          free the space used to store the number of requests
                                                   in the traffic unit;
12.13.7.5.7       else
12.13.7.5.8          TransactionDone := 'false';
12.13.7.5.9    end if
12.13.7.5.10 end proc

end module



module PP-AGGR-OP
   programs  SAVE
      /* there will be other procedure(s) in this module, e.g., a    */
      /* procedure that returns the aggregate operators for a request*/
   datasets  AggregateBuffer
                     /* used to save AggregateOperatorsMessagePtr´s */

12.9.1 proc SAVE ( input: AggregateOperatorsMessagePtr, output: nothing );
          /* This routine saves AggregateOperatorsMessagePtr to be used */
          /* by AGGREGATE-POST-OPERATION later.                         */
          /*                                                            */
          /* AggregateOperatorsMessagePtr :                             */
          /*    a pointer to ((request-id,(attribute,                   */
          /*                       aggregate-operator)[,...])[,...])    */
12.9.2 end proc

end module



module PP-RESULTS
   programs  STORE
       /* there will be other procedure(s) in this module, e.g., a  */
       /* procedure that returns the results stored in ResultsBuffer*/
   datasets  ResultsBuffer
                   /* used to store results returned by backends  */

12.13.9.1 proc STORE ( input: request-id,routing-indicator,Results,
                      output: BufferFull,last );
             /* This routine stores the results returned by a backend in */
             /* a buffer (the results will be used by REPLY-MONITOR  .   */
             /* later). It will set ´BufferFull´ to indicate whether or  */
             /* not the buffer used for storing the results returned by  */
             /* the backends is full.  It will also set ´last´ to        */
             /* indicate whether or not all the backends have returned   */
             /* their results.                                           */
             /*                                                          */
             /* request-id :                                             */
             /*    consists of (traffic-id,request-no) which uniquely    */
             /*    identifies each request being processed by MDBS       */
             /* routing-indicator :                                      */
             /*    indicates the type of results                         */
             /* Results :                                                */
             /*    results retuned from a backend                        */
             /* BufferFull :                                             */
             /*    a flag (´true´ or ´false´); it indicates whether or   */
             /*    not the buffer used for storing the results returned  */
             /*    by the backends is full                               */
             /* last :                                                   */
             /*    a flag (´true´ or ´false´); it indicates whether or   */
             /*    not all the backends have returned their results      */
             /* NoBackends :                                             */
             /*    total number of backends in MDBS. (This is a variable */
             /*    defined in SYSGEN.)                                   */

12.13.9.2    store the results into ResultsBuffer;
12.13.9.3    set the flag ´BufferFull´;
```

```
12.13.9.4    set the flag ´last´;
12.13.9.5 end proc

end module


19. end task
```

APPENDIX C

THE SSL SPECIFICATION FOR RECORD PROCESSING

The SSL specification for record processing is given in this appendix. The specification consists of five parts: a control subfunction of record processing, a retrieve processing subfunction, an insert processing subfuction, a delete processing subfunction and an update processing subfunction. They are specified in Parts I, II, III, IV and V respectively.

C.1 Part I-The Control Subfunction of Record Processing

```
/*   (1) Part I    : The control subfunction of Record Processing      */
/*   (2) Design    : RECORD PROCESSING                                 */
/*   (3) Designers : He Xingui, Masanobu Higashida                     */
/*   (4) Date      : Jan. 28, 1982                                     */
/*   (5) Modified  : Feb. 1,  1982                                     */
/*                   Feb. 18, 1982                                     */
/*                   Mar. 11, 1982                                     */
/*                   April 1, 1982                                     */
/*                   April 9, 1982                                     */
/*                   April 15,1982                                     */
/*                   April 27,1982                                     */
/*                   May  17, 1982                                     */
/*                   May  19, 1982                                     */
/*   (6) Purpose   :                                                   */
/*       The control subfunction of Record Processng is designed for   */
/* analyzing the information coming from Directory Management to decide*/
/* what request processing subfunction should be executed, and then to */
/* transfer control to the relevant procedure.                         */
/*   (7) Input:                                                        */
/*       Input consists of a formated request and a set of disk        */
/* addresses where the relevant data are stored.                       */
/*   (8) Procedure Hierarchy for The Control Subfunction               */

                Control Subfunction of Record Processing
                                  |
        +----------------+--------+---------+----------------+
        |                |                  |                |
        |                |                  |                |
    Retrieve          Insert             Delete           Update
    Processing        Processing         Processing       Processing
    Subfunction       Subfunction        Subfunction      Subfunction
```

```
/*   (9) Jackson Chart:                                                */

                    +------------------------+
                    | Control Subfunction of |
                    | Record Processing      |
                    +------------------------+
                            |    |   |
        +-------------------+    |   +----------------------+
        |                        |                          |
 +----------------+        +-----------------+   +------------------------+
 | Get an item in |        | Process a       |   | Inform CONTROLLER      |
 | a queue formed |        | request waiting |   | the processing finished|
 | by DMT         |        | in the queue    |   +------------------------+
 +----------------+        +-----------------+
                            |  |     |  |
          +-----------------+  |     |  +-----------------+
          |               +----+     +----+               |
          |               |               |               |
    +------------+  +------------+  +------------+  +------------+
    | Retrieve |o|  | Update   |o|  | Delete   |o|  | Insert   |o|
    | request  +-|  | request  +-|  | request  +-|  | request  +-|
    | processing |  | processing |  | processing |  | processing |
    +------------+  +------------+  +------------+  +------------+
/*   (11) Program Specifications                                          */


task RECORD_PROCESSING;

/* This task is to be used to analyze a request and then execute    */
/* the  relevant procedure.                                          */


list REQUEST: string;
set ADDRESSES: integer;
scalar NewTrack: logical;

while 'true' do              /* Do forever.                          */
 perform GET_REQ_ADD_NEW(REQUEST,ADDRESSES,NewTrack);
                             /* Get a message( a request REQUEST,    */
                             /* a set of addresses ADDRESSES         */
                             /* and new track indicator NewTrack)    */
                             /* from a queue.                        */
 case REQUEST.REQUEST_TYPE value

  'RETRIEVE':
       perform RETRIEVE_PROCESSING(REQUEST.QUERY,
                                      REQUEST.TARGET, ADDRESSES);
  'UPDATE':
       perform UPDATE_PROCESSING(REQUEST.QUERY, ADDRESSES,
                                 REQUEST.attribute,REQUEST.valuel);
  'DELETE':
       perform DELETE_PROCESSING(REQUEST.QUERY, ADDRESSES);
  'INSERT':
       perform INSERT_PROCESSING(REQUEST.RECORD, ADDRESSES,NewTrack);

 end case;

 perform SEND_MESSAGE  /* Send completion signal  to CONTROLLER.    */

end while;
```

C.2 Part II-The Retrieve Processing Subfunction

```
/*   (1) Part II   : Retrieve Processing Subfunction                   */
/*   (2) Design    : RECORD PROCESSING                                 */
/*   (3) Designers : He Xingui, Masanobu Higashida                     */
/*   (4) Date      : Jan. 28, 1982                                     */
/*   (5) Modified  : Feb. 1,  1982                                     */
/*                   Feb. 18, 1982                                     */
/*                   Mar. 11, 1982                                     */
/*                   April 1, 1982                                     */
/*                   April 9, 1982                                     */
/*                   April 15,1982                                     */
/*                   April 27,1982                                     */
/*                   May  17, 1982                                     */
/*                   May  19, 1982                                     */
/*   (6) Purpose   :                                                   */
/*       The procedure is designed for retrieving the records          */
/* satisfying the query in the request.                                */
/*   (8) Procedure Hierarchy for Retrieve Processing Subfunction        */


                    Retrieve Processing Subfunction
                                   |
    +------------+------------+----+----+---------+----------+----------+
    |            |            |         |         |          |          |
GET_BUFFER       |      CHECK_QUERY     |      PROJECT       |        FLUSH_
                 |                      |                    |        BUFFER
            FETCH_TO_              AGGREGATE              STUFF_
            TRACK_BUFFER           OPERATION              BUFFER
```

```
/*   (9) Jackson Chart:                                               */

                            +------------+
                            | Retrieve   |
                            | request    |
                            | processing |
                            +------------+
                             |   |   |
        +--------------------+   |   +--------------+
        |                        |                  |
  +-------------+        +------------+    +----------------+
  | Get RESULT_ |        |Retrieve    |    | Flush RESULT_  |
  | BUFFER      |        | results    |    | BUFFER         |
  +-------------+        +------------+    +----------------+
                          |   |    |
        +-----------------+   |    +-------------------------+
        |                     |                              |
+----------------+     +--------------------+   +----------------+
| Initialize     |     | Process data     |*|   |Stuff           |
| partial results|     | track by track   +-|   |partial results |
+----------------+     +--------------------+   +----------------+
                           |            |
                +----------------+   +------------+
                | Fetch one track|   | Retrieve   |
                | to TRACK_BUFFER|   | records    |
                +----------------+   +------------+
                                            |
                                  +---------------------+
                                  | Select  records   |*|
                                  | in TRACK_BUFFER   +-|
                                  | one by one          |
                                  +---------------------+
                                            |
                                  +---------------------+
                                  | Check whether the   |
                                  | record has been     |
                                  | marked for deletion |
                                  +---------------------+
                                       |             |
                              +--------------+ +----------------+
                              | Deleted    |o| | Not deleted  |o|
                              +--------------+ +----------------+
                                |                   |
                     +-------------+ +---------------------+
                     | Do nothing  | | Check whether the |*|
                     +-------------+ | record satisfies  +-|
                                     | the query           |
                                     +---------------------+
                                       |            |
                       +---------------+            |
                       |                            |
                +--------------+        +---------------------+
                | Satisfied  |o|        |When not satisfied |o|
                | processing +-|        |                   +-|
                +--------------+        +---------------------+
                   |       |                        |
           +-------+       +------+           +------------+
           |                      |           | Do nothing |
     +-------------+         +-------------+  +------------+
     | Collect   |o|         | Aggregate |o|
     | the       +-|         | operation +-|
     | results     |         | and collect |
     | into RESULT_|         | the results |
     | BUFFER      |         | into RESULT_|
     +-------------+         | BUFFER      |
           |                 +-------------+
           |                        |
      +----------+            +----------+
      |Stuff into|            |Stuff into|
      |the buffer|            |the buffer|
      +----------+            +----------+
  +-------------------+
  | Check whether the |
  | record satisfies  |
  | the query         |
  +-------------------+
            |
+---------------------+
| Check the record  |*|
| with each         +-|
| conjunction         |
+---------------------+
            |
+---------------------+
| Check the record  |*|
| with each         +-|
| predicate  in       |
| the conjunction     |
+---------------------+
/*   (11) Program Specifications                                          */

8.1  proc RETRIEVE_PROCESSING(input: QUERY,TARGET,ADDRESSES);
    /* This procedure is to be used for processing of RETRIEVE request.   */

8.2  list QUERY,TARGET,result: string;
8.3  set ADDRESSES: integer;
8.4  array TRACK_BUFFER,RESUT_BUFFER: word;
8.5  scalar indexA,indexB: integer;
      /* these are pointers for ADDRESSES and TRACK_BUFFER respectively   */
8.6  scalar satisfied, ok : logical;
8.7  scalar sum,count,max,min: real;

8.8  perform GET_BUFFER(RESULT_BUFFER);           /* Get RESULT_BUFFER.    */
      /* Initialize the partial results.                                  */
8.9  sum:=0;
8.10 count:=0;
8.11 max:=the smallest number;
8,12 min:=the largest number;

      /* Process data track by track .                                    */
8.13 for each address ADDRESSES(indexA) in ADDRESSES do

      /* Fetch one track into TRACK_BUFFER.                               */
8.14   perform FETCH_TO_TRACK_BUFFER(indexA,TRACK_BUFFER);

      /* Select records in TRACK_BUFFER one by one.                       */
8.15  for each record TRACK_BUFFER(indexB) in TRACK_BUFFER do
8.16    if the record is not marked a ´deletion flag´
8.17     then
          /* Check whether the record satisfies the QUERY.                */

8.18       perform CHECK_QUERY(QUERY,TRACK_BUFFER,indexB,satisfied);

8.19       if satisfied=´true´
8.20        then
8.21          if there is aggregate operation
8.22              then
                          /* Compute partial results and count.          */
8.23                perform AGGREATE_OPERATION(TRACK_BUFFER,indexB,
                                             sum,count,max,min,TARGET);
8.24              else
                          /* Get result by projection.                   */
8.25                perform PROJECT(TRACK_BUFFER,indexB,result,TARGET);

                          /* Collect it into RESULT_BUFFER.              */
8.26                perform STUFF_BUFFER(RESULT_BUFFER,
                                            result,length_of_result);
8.27          end if;
8.28       end if;
8.29    end if;
8.30  end for; /* indexB */
8.31 end for;   /* indexA */
```

```
                /* Stuff the partial results into RESULT_BUFFER, if any.      */
8.32 perform STUFF_BUFFER(RESULT_BUFFER, sum, length_of_sum);
8.33 perform STUFF_BUFFER(RESULT_BUFFER, count, length_of_count);
8.34 perform STUFF_BUFFER(RESULT_BUFFER, max, length_of_max);
8.35 perform STUFF_BUFFER(RESULT_BUFFER, min, length_of_min);

                /* Send the collected results in RESULT_BUFFER to CONTROLLER. */
8.36 perform FLUSH_BUFFER(RESULT_BUFFER, ok );
8.37 end proc;


8.18.1 proc CHECK_QUERY(input: QUERY,TRACK_BUFFER,indexB, output:satisfied);
              /* This procedure is used to check whether the record in        */

              /* TRACK_BUFFER(indexB)  satisfies the QUERY or not.            */

8.18.2  list QUERY: string;
8.18.3  array TRACK_BUFFER: word;
8.18.4  scalar indexB,indexC,indexP: integer;
           /* these are pointers for TRACK_BUFFER, QUERY.CONJUNCTION          */
           /* and QUERY.CONJUNCTION.PREDICATE respectively.                   */
8.18.5  scalar satisfied: logical;

           /* Check whether the record satisfies the QUERY                    */
8.18.6  for each conjunction QUERY(indexC,*) in QUERY do
          satisfied='true';
           /* Check whether the record satisfies the conjunction              */
           /* pointed by indexC.                                              */
8.18.7    for each predicate QUERY(indexC,indexP) in QUERY(indexC,*) do
8.18.8      if The record in TRACK_BUFFER(indexB) does not satisfy
                           the predicate in QUERY(indexC,indexP);
8.18.9       then
8.18.10           satisfied:='false';
8.18.11           exit the loop;
8.18.12     end if;
8.18.13   end for;    /* indexP */
8.18.14   if satisfied:='true' then
8.18.15     return ;
8.18.16   end if;
8.18.17 end for;      / *indexC */
8.18.18 end proc;
```

C.3 part III-The Insert Processing Subfunction

```
/*   (1) Part III   : Insert Processing Subfunction                     */
/*   (2) Design     : RECORD PROCESSING                                 */
/*   (3) Designers  : He Xingui, Masanobu Higashida                     */
/*   (4) Date       : Jan. 28, 1982                                     */
/*   (5) Modified   : Feb. 1,  1982                                     */
/*                    Feb. 18, 1982                                     */
/*                    Mar. 11, 1982                                     */
/*                    April 1, 1982                                     */
/*                    April 9, 1982                                     */
/*                    April 15,1982                                     */
/*                    April 27,1982                                     */
/*                    May  17, 1982                                     */
/*                    May  19, 1982                                     */
/*   (6) Purpose    :                                                   */
/*       The procedure is designed for inserting a record into the disk*/
/* indicated by a address.                                              */
```

```
(8) Procedure Hierarchy for Insert Processing Subfunction          */



                   Insert processing Subfunction
                                  |
       +--------------------------+--------------------------+
       |                          |                          |
FETCH_TO_TRACK_BUFFER        INSERT RECORD          STORE_TRACK_BUFFER
```

```
/*   (9) Jackson Chart:                                                 */



                           +-------------+
                           | Insert      |
                           | request     |
                           | processing  |
                           +-------------+
                             |  |   |
           +-----------------+  |   |
           |                    |   +----------+
           |                    |              |
  +----------------+ +---------------+ +--------------------+
  |Should it insert| | Insert record | | Store TRACK_BUFFER |
  |into a new track| | into TRACK_   | | back to disk       |
  +----------------+ | BUFFER        | +--------------------+
     |            |  +---------------+
  +-------+    +-------+
  | No  |o|    | Yes |o|
  +-------+    +-------+
     |            |
+----------+-+ +----------+-+
|Fetch one   | |Do nothing  |
|track to    | +------------+
|TRACK_BUFFER|
+------------+
/*   (11) Program Specifications                                         */

11.1  proc INSERT_PROCESSING(input: RECORD,ADDRESS,NewTrack);
    /* This procedure is used for inserting the record into TRACK_BUFFER  */
    /* according to ADDRESS.                                              */

11.2    set ADDRESS: integer;
11.3    array TRACK_BUFFER: word;
11.4    scalar RECORD: string;
11.5    scalar NewTrack: logical;

11.6    if NewTrack='false'
11.7     then
           /* Fetch one track indicated by ADDRESS to TRACK_BUFFER.       */
11.8       perform FETCH_TO_TRACK_BUFFER(ADDRESS,TRACK_BUFFER);
11.9    end if;

           /* Insert the record into the TRACK_BUFFER.                    */
11.10      perform INSERT_RECORD(RECORD,TRACK_BUFFER);

           /* Store TRACK_BUFFER back to the disk according to ADDRESS.   */
11.11      perform STORE_TRACK_BUFFER(ADDRESS,TRACK_BUFFER);
11.12 end proc;
```

C.4 Part IV-The Update processing Subfunction

```
/*   (1) Part IV    : Update processing Subfunction                   */
/*   (2) Design     : RECORD PROCESSING                               */
/*   (3) Designers  : He Xingui, Masanobu Higashida                   */
/*   (4) Date       : Jan. 28, 1982                                   */
/*   (5) Modified   : Feb. 1,  1982                                   */
/*                    Feb. 18, 1982                                   */
/*                    Mar. 11, 1982                                   */
/*                    April 1, 1982                                   */
/*                    April 9, 1982                                   */
/*                    April 15,1982                                   */
/*                    April 27,1982                                   */
/*                    May  17, 1982                                   */
/*                    May  19, 1982                                   */
/*   (6) Purpose    :                                                 */
/*       The procedure is designed for implementing the update request */
/* with the modifier of type 0, type I or type II. the request with    */
/* modifier of type III or IV is implemented as a retrieve followed by */
/* another type 0 update.                                             */
/*   (8) Procedure Hierarchy for Update Processing Subfunction         */

                    Update Processing Subfunction
                                   |
       +----------+----------------+--------+--------+------------+
       |          |                |        |        |            |
GET_BUFFER   FETCH_TO       CHECK_QUERY   UPDATE   STORE_   FLUSH_BUFFER
             TRACK_BUFFER                   |      TRACK_
                                            |      BUFFER
                                            |
   +---------------------+------------------+-----------+----------------+
   |                     |                  |           |                |
GET_ATTRIBUTE_    UPDATE_RECORD    CHECK_CLUSETR   STUFF_BUFFER       DELETE_
VALUE                                                                 RECORD
```

```
/*   (9) Jackson Chart:                                                */

          +---------------------------+
          | Update request Processing |
          +---------------------------+
            |          |           |
+-------------+   +--------+    +----------------+
| Get RESULT_ |   | Update |    | Flush RESULT_  |
| BUFFER      |   | records|    | BUFFER         |
+-------------+   +--------+    +----------------+
                       |
             +-------------------+
             | Process data    |*|
             | track by track  +-|
             +-------------------+
              |        |        |
+---------------+ +--------+ +------------+
|Fetch one track| |Update  | |Store TRACK_|
|to TRACK_BUFFER| |records | |BUFFER      |
+---------------+ +--------+ |back to disk|
                       |     +------------+
                       |
                  +---------------------+
                  | Select  records   |*|
                  | in TRACK_BUFFER   +-|
                  | one by one          |
                  +---------------------+
                                     |
                                  +---------------------+
                                  | Check if the record |
                                  | has been marked for |
                                  | deletion            |
                                  +---------------------+
                                   |                   |
                           +------------+     +---------------+
                           | Deleted  |o|     | Not deleted |o|
                           +------------+     +---------------+
                               |                       |
                         +------------+ +---------------------+
                         | Do nothing | | Check whether the |*|
                         +------------+ | record satisfies  +-|
                                        | the query           |
                                        +---------------------+
                                            |             |
                   +-------------------------+  +---------------------+
                   | Satisfied processing |o|  |When not satisfied |o|
                   +-------------------------+  +---------------------+
                    |         |            |
           +---------+  +------------+  +-----------------+
           | Compute |  | Update the |  | Check if change |
           | valuel  |  | record in  |  | cluster or not  |
           |         |  |TRACK_BUFFER|  |                 |
           +---------+  +------------+  +-----------------+
                                         |             |
                       +-------------------+ +---------------------+
                       | It needs to     |o| | It does not need  |o|
                       | change cluster  +-| | to change cluster +-|
                       +-------------------+ +---------------------+
                         |               |               |
              +-------------+  +-----------------+  +-------------+
              | Collect the |  | Delete original |  |             |
              | record into |  | record in the   |  | Do nothing  |
              |RESULT_BUFFER|  | TRACK_BUFFER    |  |             |
              +-------------+  +-----------------+  +-------------+
                     |
                +----------+
                |stuff into|
                |the buffer|
                +----------+
/*   (11) Program Specifications                                         */

9.1  proc UPDATE_PROCESSING(input; QUERY,ADDRESSES,attribute,valuel);
    /* This procedure is to be used for processing of UPDATE  request. */

9.2  list QUERY: string;
9.3  set ADDRESSES: integer;
9.4  array TRACK_BUFFER,RESULT_BUFFER: word;
9.5  scalar attribute,valuel: string;
9.6  scalar indexA indexB: integer;
      /* These are  pointers for ADDRESSES and TRACK_BUFFER              */
      /* respectively.                                                   */
9.7  scalar satisfied, ok: logical;

9.8  perform GET_BUFFER(RESULT_BUFFER);          /* Get RESULT_BUFFER.  */

      /* Process data track by track .                                   */
9.9 for each address ADDRESSES(indexA) in ADDRESSES do

      /* Fetch one track into TRACK_BUFFER.                              */
9.10   perform FETCH_TO_TRACK_BUFFER(indexA,TRACK_BUFFER);

      /* Select records in TRACK_BUFFER one by one.                      */
9.11   for each record TRACK_BUFFER(indexB) in TRACK_BUFFER do
9.12     if the record is not marked a ´deletion flag´
9.13       then
             /* Check whether the record satisfies the QUERY.            */
9.14        perform CHECK_QUERY(QUERY,TRACK_BUFFER,indexB,satisfied);

9.15        if satisfied=´true´
9.16         then        /* Update the retrieved record in               */
                         /* TRACK_BUFFER(indexB) and collect it into     */
                         /* RESULT_BUFFER, or store it back to           */
                         /* the original place.                          */
9.17          perform UPDATE(indexB,attribute,valuel,
                                      TRACK_BUFFER,RESULT_BUFFER);
9.18        end if;
9.19     end if;
9.20   end for; /* indexB */

       /* Store TRACK_BUFFER back to disk.                               */
9.21   perform STORE_TRACK_BUFFER(indexA,TRACK_BUFFER);
9.22 end for;    /* indexA */

     /* Send the collected results in RESULT_BUFFER to CONTROLLER.       */
9.23 perform FLUSH_BUFFER(RESULT_BUFFER, ok);
9.24 end proc;
```

```
9.17.1  proc UPDATE(indexB,attribute,valuel,TRACK_BUFFER,RESULT_BUFFER);

      /* This procedure updates the record in TRACK_BUFFER(indexB).     */

9.17.2  array TRACK_BUFFER,RESULT_BUFFER: word;
9.17.3  scalar attribute,value,valuel: string;
9.17.4  scalar indexB: integer;  /*  This is a pionter for TRACK_BUFFER */
9.17.5  scalar check_result: logical;

       /* Get the attribute value of the record in TRACK_BUFFER(indexB).*/
9.17.6   perform GET_ATTRIBUTE_VALUE(indexB,attribute,value);
       /* Here f(...) is a function procedure.                          */
9.17.7   valuel=f(value);

       /* Update the value of attribute in TRACK_BUFFER(indexB)         */
       /* into valuel.                                                  */
9.17.8   perform UPDATE_RECORD(attribute,valuel,TRACK_BUFFER,indexB);

         /* Check if the record needs to change cluster or not.         */
9.17.9   perform CHECK_CLUSTER(TRACK_BUFFER,indexB,cluster_changed);

9.17.10     if cluster_changed='true'
9.17.11      then  /* The updated record needs to change cluster.       */
                   /* Collect the record into RESULT_BUFFER.            */
9.17.12          perform STUFF_BUFFER(RESULT_BUFFER,TRACK_BUFFER(indexB),
                                                   length_of_the_record);

                   /* Delete the original record in TRACK_BUFFER.       */
9.17.13          perform DELETE_RECORD(TRACK_BUFFER,indexB);
9.17.14     end if;
9.17.15 end proc;
```

C.5 Part V-The Delete Processing Subfunction

```
/*   (1) Part V    : Delete Processing Subfunction                  */
/*   (2) design    : RECORD PROCESSING                              */
/*   (3) Designers : He Xingui, Masanobu Higashida                  */
/*   (4) Date      : Jan. 28, 1982                                  */
/*   (5) Modified  : Feb. 1,  1982                                  */
/*                   Feb. 18, 1982                                  */
/*                   Mar. 11, 1982                                  */
/*                   April 1, 1982                                  */
/*                   April 9, 1982                                  */
/*                   April 15,1982                                  */
/*                   April 27,1982                                  */
/*                   May  17, 1982                                  */
/*                   May  19, 1982                                  */
/*   (6) Purpose   :                                                */
/*       The procedure is designed for deleting the records satisfying */
/* the query in the request.                                        */
/*   (8) Procedure Hierarchy for Delete Processing Subfunction        */


                  Delete Procesing Subfunction
                               |
     +----------------+--------+--------+----------------+
     |                |                 |                |
     |                |                 |                |
FETCH_TO        CHECK_QUERY          DELETE       STORE_TRACK_BUFFER
TRACK_BUFFER
```

```
/*   (9) Jackson Chart:                                               */

                +-----------+
                | Delete    |
                | request   |
                | processing|
                +-----------+
                      |
             +-------------------+
             | Process data    |*|
             | track by track  +-|
             +-------------------+
                |      |       |
                |      |       +---------------+
                |      +-----+                 |
                |            |                 |
   +----------------+   +-------------+  +--------------------+
   | Fetch one track|   | Delete(mark)|  | Store TRACK_BUFFER |
   | to TRACK_BUFFER|   | records     |  | back to disk       |
   +----------------+   +-------------+  +--------------------+
                               |
                        +---------------------+
                        | Select  records    |*|
                        | in TRACK_BUFFER    +-|
                        | one by one           |
                        +---------------------+
                                           |
                                  +---------------------+
                                  | Check if the record |
                                  | has been marked for |
                                  | deletion            |
                                  +---------------------+
                                    |                 |
                              +-----------+    +---------------+
                              | Deleted |o|    | Not deleted |o|
                              +-----------+    +---------------+
                               |                      |
                         +-----------+  +---------------------+
                         | Do nothing|  | Check whether the |*|
                         +-----------+  | record satisfies  +-|
                                        | the query           |
                                        +---------------------+
                                           |                |
                                  +--------------+  +---------------------+
                                  | Satisfied   |o|  |When not satisfied |o|
                                  | processing  +-|  |                   +-|
                                  +--------------+  +---------------------+
                                           |                   |
                                   +---------------+     +------------+
                                   | Mark the      |     | Do nothing |
                                   | record  in    |     +------------+
                                   | TRACK_BUFFER  |
                                   +---------------+
/*   (11) Program Specifications                                          */

proc DELETE_PROCESSING( input: QUERY,ADDRESSES);
    /* This procedure is to be used for processing of DELETE request.     */

list QUERY: string;
set ADDRESSES: integer;
array TRACK_BUFFER: word;
scalar indexA,indexB: integer;
/*These are pointers for ADDRESSES and TRACK_BUFFER respectively         */
scalar satisfied: logical;

/* Process data track by track .                                          */

for each address ADDRESSES(indexA) in ADDRESSES 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 a ´deletion flag´
      then
      /* Check whether the record satisfies the QUERY.                    */

       perform CHECK_QUERY(QUERY,TRACK_BUFFER,indexB,satisfied);
       if satisfied=´true´
        then   /* Mark the retrieved record in TRACK_BUFFER(indexB).*/
         perform DELETE(TRACK_BUFFER,indexB);
       end if;
    end if;
  end for; /* indexB */
  pertorm STORE_TRACK_BUFFER(indexA,TRACK_BUFFER);
                 /* Store TRACK_BUFFER back to disk.                      */
end for;    /* indexA */
end proc;
```

APPENDIX D

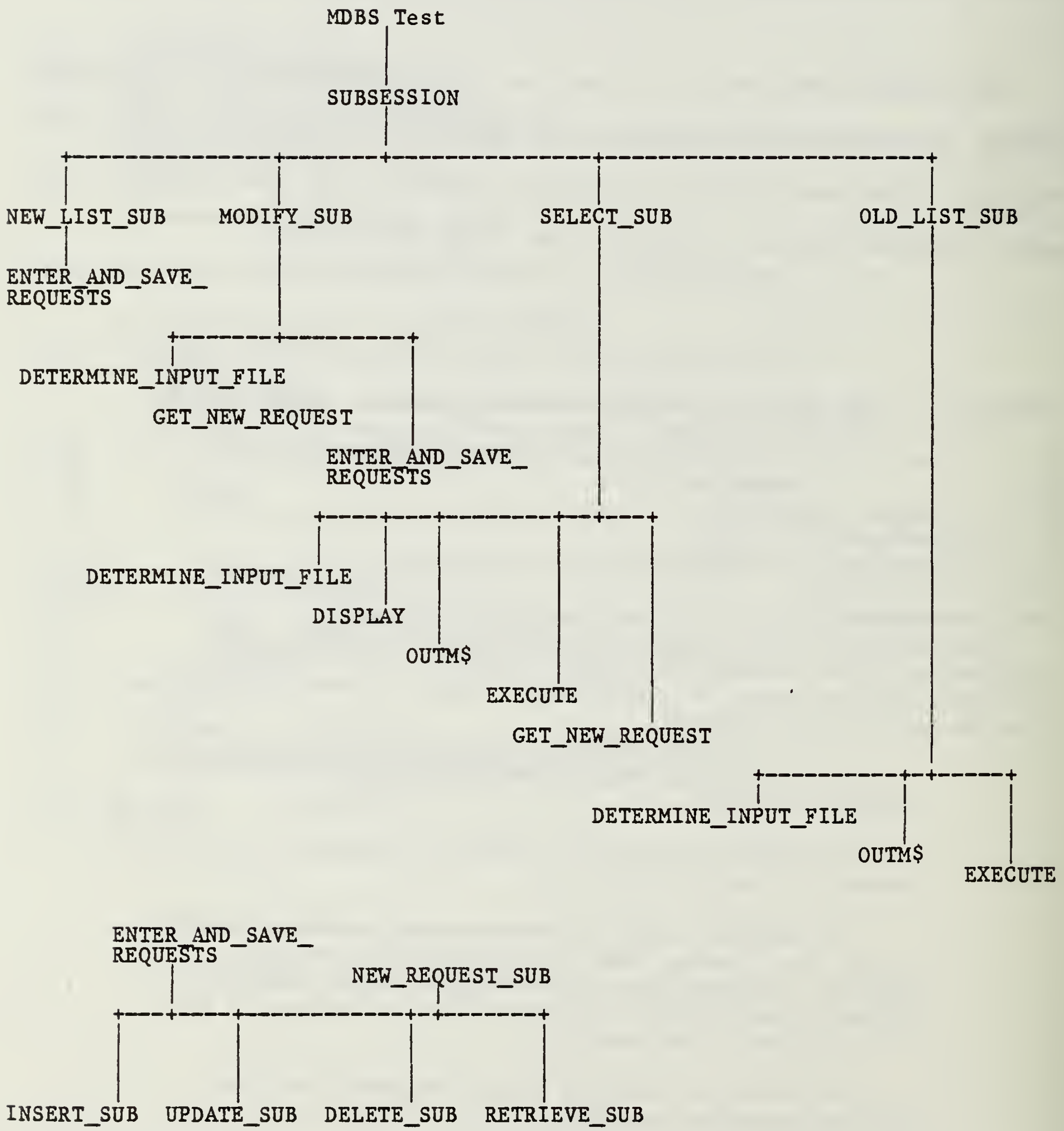
THE SSL SPECIFICATION FOR THE TEST REQUEST GENERATION AND EXECUTION PACKAGE

The program specification for the test request generation and execution package is shown in this appendix. The specification design is composed of two parts. Part one includes the top level design. Part two includes the module concerned with the handling of the output from a test.

D.1 Part I - The Top Level of Test Request Generation and Execution

```
/*   (1) Part I -  The Top Level of Test Request Generation and              */
/*                  Execution                                                 */
/*   (2) Design: MDBS TEST                                                    */
/*   (3) Designer: D.S. Kerr                                                  */
/*   (4) Date: July 8, 1982                                                   */
/*                                                                            */
/*   (6) Purpose:                                                             */
     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 executed one at a time.  A request
  is sent to MDBS.  Then the program waits for a response before sending
  the next request.  Transactions are not allowed.

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

(8) Procedure Hierarchy for MDBS Test



(10) Data Structures

The data structures definitions are included at the beginning of each procedure definition in (11) below.

(11) Program Specifications

```
1.        task MDBS Test;
2.        scalar more-subsessions; /* flag: TRUE - continue, FALSE - stop */

3.            Print initial message to user;
4.            more-subsessions := TRUE;
5.            while more-subsessions do
6.                perform SUBSESSION;
7.                Prompt for continue message;
8.                Read continue message;
9.                if user does not want to continue
10.                   then
11.                       more-subsessions := FALSE;
12.               end if
13.           end while;
14.       end task;



6.        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)        */

6.1       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.        */
6.2       scalar type-of-subsession; /* Possible values are NEW_LIST,
                  MODIFY, SELECT and OLD_LIST */

6.3           Prompt for next type-of-subsession;
6.4           Read next type-of-subsession;
6.5           case type-of-subsession value
6.6               NEW_LIST: /* Enter a new request-list */
                      perform NEW_LIST_SUB( current-request-file);
6.7               MODIFY: /* Modify an old list */
                      perform MODIFY_SUB( current-request-file );
6.8               SELECT: /* Select requests, one at a time, from an */
                          /* existing request-list */
                      perform SELECT_SUB( current-request-file );
6.9               OLD_LIST: /* Execute an existing request-list */
                      perform OLD_LIST_SUB( current-request-file);
```

```
6.10              otherwise: Print error message;
6.11          end case;
6.12      end procedure;



6.6.1     procedure NEW_LIST_SUB( output: current-request-file );
6.6.2         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.                                                */

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

6.6.6         Prompt for request-list-file-name;
6.6.7         Read request-list-file-name;
6.6.8         Open file( request-list-file-name ) output;
6.6.9         perform ENTER_AND_SAVE_REQUESTS( request-list-file-name );
6.6.10        Close file( request-list-file-name );
6.6.11        current-request-file := request-list-file-name;
6.6.12    end procedure;



6.7.1     procedure MODIFY_SUB( input/output: current-request-file );
6.7.2         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.                                                */

6.7.3     scalar input-request-file; /* The list of requests to be modified. */
6.7.4     scalar new-request-file; /* The new list of requests. */
6.7.5     scalar next-version; /* flag: TRUE - set new-request-file to next  */
                  /* version of input-request-file, FALSE - get new name.   */
```

```
6.7.6     record request;
6.7.7     scalar more-requests-in-input-request-file; /* continuation flag */
6.7.8     scalar more-requests-to-enter; /* continuation flag */
6.7.9     scalar change-type; /* ADD, MODIFY, REMOVE, or NOCHANGE */

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

              /* Determine input-request-file to be modified. */
6.7.11        perform DETERMINE_INPUT_FILE( current-request-file,
                                          input-request-file );
6.7.12        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.*/
6.7.13        Prompt user to determine next-version;
6.7.14        Read next-version;
6.7.15        if next-version
6.7.16           then
6.7.17               Set new-request-file to next version of
                                          input-request-file;
6.7.18           else
6.7.19               Prompt for new-request-file name;
6.7.20               Read name of new-request-file;
6.7.21        end if;
6.7.22        open file( new-request-file ) output;

6.7.23        Read first request from input-request-file;
6.7.24        more-requests-in-input-request-file := TRUE;

6.7.25        while more-requests-in-input-request-file do
6.7.26           Prompt user for change-type for this request;
6.7.27           Read change-type;
6.7.28           case change-type value
6.7.29               ADD: /* enter and save the next request */
                         perform GET_NEW_REQUEST( request );
6.7.30                   Write request into new-request-file;
6.7.31               MODIFY:
                         Prompt and get modified request from user;
6.7.32                   Write new request into new-request-file;
6.7.33                   Read next request from input-request-file;
6.7.34               REMOVE:
                         Read next request from input-request-file;
6.7.35               NOCHANGE:
                         Write current request into new-request-file;
6.7.36                   Read next request from input-request-file;
6.7.37               otherwise: Print system error message;
6.7.38           end case;
6.7.39        end while;

              /* Note that at this point all the old requests have been    */
              /* processed.  However it is possible that the user wants to */
              /* append more requests.                                     */
6.7.40        Prompt user that input file has been processed, but that more
                 requests may still be appended;
```

```
6.7.41        perform ENTER_AND_SAVE_REQUESTS( new-request-file );
6.7.42        close file( input-request-file );
6.7.43        close file( new-request-file );
6.7.44        current-request-file := new-request-file;
6.7.45    end procedure;


6.8.1     procedure SELECT_SUB( input/output: current-request-file );
6.8.2         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.          */

6.8.3     scalar input-request-file; /* The file containing the requests. */
6.8.4     array requests( MAX_NUMBER_OF_REQUESTS );
                          /* from input-request-file */
6.8.5         scalar number-of-requests; /* The actual number in */
                  /* input-request-file must be less than        */
                  /* MAX_NUMBER_OF_REQUESTS                      */
6.8.6         scalar request-number; /* of the request chosen */
6.8.7     record new-request; /* Provided by user. */
6.8.8     record response; /* to the request being executed. */

6.8.9     scalar more-to-execute; /* flag to control loop */
6.8.10    scalar next-operation; /* Values can be REQUEST_NUMBER, DISPLAY, */
                  /* NEW_REQUEST or STOP                                   */

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

              /* Determine whether response is to go to CRT, file or both. */
6.8.16        perform OUTM$FORMAT;
6.8.17        more-to-execute := TRUE;

6.8.18        while more-to-execute do
6.8.19            Prompt user for next-operation /* It should be either a */
                          /* request-number, a request-to-display or a    */
                          /* new-request                                  */
6.8.20            Read next-operation;
6.8.21            case next-operation value
6.8.22                REQUEST_NUMBER:
                          Check that request-number is less than
                                    number-of-requests;
```

```
6.8.23                   perform EXECUTE( requests(request-number),
                                 response );
                         /* Output the response to CRT, file or CRT&file,
                                 as appropriate. */
6.8.24                   perform OUTM$RESPONSE( response );

6.8.25               DISPLAY: perform DISPLAY( requests );
6.8.26               NEW_REQUEST:
                         perform GET_NEW_REQUEST( new-request );
6.8.27                   perform EXECUTE( new-request, response );
                         /* Output the response to CRT, file or CRT&file,
                                         as appropriate. */
6.8.28                   perform OUTM$RESPONSE( response );

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

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

6.8.35   end procedure;



6.9.1    procedure OLD_LIST_SUB( current-request-file );
6.9.2        scalar current-request-file; /* The name of the file */

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

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

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

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

6.9.15       perform OUTM$FINISH;
```

```
6.9.16        close( input-request-file );
6.9.17        current-request-file := input-request-file;

6.9.18    end procedure;



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

6.6.9.5       next-step := I;
6.6.9.6       while next-step ~= F do
6.6.9.7           Prompt for next-step;
6.6.9.8           case next-step value
6.6.9.9               I: /* enter and save the next insert request */
                          perform INSERT_SUB( request );
6.6.9.10                  Write request into request-list-file-name ;
6.6.9.11              R: /* enter and save the next retrieve request */
6.6.9.12                  perform RETRIEVE_SUB( request );
6.6.9.13                  Write request into request-list-file-name ;
6.6.9.14              U: /* enter and save the next update request */
6.6.9.15                  perform DELETE_SUB( request );
6.6.9.16                  Write request into request-list-file-name ;
6.6.9.17              D: /* enter and save the next delete request */
6.6.9.18                  perform DELETE_SUB( request );
6.6.9.19                  Write request into request-list-file-name ;
6.6.9.20              F: /* Finish entering requests */
6.6.9.21              otherwise: Print error message;
6.6.9.22          end case;
6.6.9.23      end while;
6.6.9.24  end procedure;



6.7.11.1   procedure DETERMINE_INPUT_FILE( input: current-request-file,
6.7.11.2           output: input-request-file );
6.7.11.3       scalar current-request-file;
6.7.11.4       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.                                           */

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

6.7.11.6       if current-request-file is NULL
6.7.11.7           then
6.7.11.8               Prompt for name of input-request-file;
6.7.11.9               Read name of input-request-file;
```

```
6.7.11.10        else /* Determine if user wants to use the */
                      /* current-request-file or a different old file. */
6.7.11.11            Prompt user to determine modify-current-file-flag;
6.7.11.12            Read modify-current-file-flag;
6.7.11.13            if modify-current-file-flag
6.7.11.14                then
6.7.11.15                    Prompt for name of input-request-file;
6.7.11.16                    Read name of input-request-file;
6.7.11.17                else
6.7.11.18                    input-request-file := current-request-file;
6.7.11.19            end if;
6.7.11.20    end if;
6.7.11.21 end procedure;



6.7.29.1         procedure GET_NEW_REQUEST( output: request );
6.7.29.2             record request; /* to be obtained from user */

                     /* Prompts user for information necessary to enter a  */
                     /* new request.  Returns the request.                */

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

6.7.29.4             Prompt for next request-type;
6.7.29.5             Read request-type;
6.7.29.6                 case request-type value
6.7.29.7                     I: perform INSERT_SUB( request );
6.7.29.8                     U: perform UPDATE_SUB( request );
6.7.29.9                     D: perform DELETE_SUB( request );
6.7.29.10                    R: perform RETRIEVE_SUB( request );
6.7.29.11                    otherwise: Print error message;
6.7.29.12                end case;

6.7.29.13        end procedure;



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

6.8.15.2 array requests( MAX_NUMBER_OF_REQUESTS );
                                 /* to be displayed. */

6.8.15.3 end procedure;
```

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

6.8.23.2 record request; /* to be executed */
6.8.23.3 record response; /* to the execution of the request */

6.8.23.4 end procedure;
```

D.2 Part II - The Output Module for Test Execution

```
/*   (1) Part II -  The Output Module for Test Execution                 */
/*   (2) Design: OUTM                                                     */
/*   (3) Designer: D.S. Kerr                                              */
/*   (4) Date: July 8, 1982                                               */
/*                                                                        */
/*   (6) Purpose:                                                         */
/* The following three procedures are used to handle the displaying     */
/* and/or saving of the responses to the execution of the requests.      */
/* The default is to display the responses on the CRT.                   */
```

(8) Procedure Hierarchy for the Module OUTM

```
+-------------------+
|     OUTM$         |
+-------------------+
|                   |
|     FORMAT        |
|     RESPONSE      |
|     FINISH        |
|                   |
+-------------------+
```

(10) Data Structures

The data structures definitions are included as part of the module and at the beginning of each procedure definition in (11) below.

(11) Program Specifications

```
module OUTM

    programs FORMAT, RESPONSE, FINISH;
    data sets

         /* Variables controlling the output of responses */
         scalar CRT-output-flag;
                 /* TRUE if output is to be displayed on CRT. */
                 /* Initial value is TRUE. */
         scalar file-output-flag;
                 /* TRUE if output is to be put into a file. */
                 /* Initial value is FALSE. */
         /* CRT-output-flag and/or file-output-flag must be TRUE     */
         scalar response-file-name;
                 /* the name of the file if output is to be made */
                 /* to a file.                                   */
end module;


         procedure FORMAT;
             /* Determines what form of output is to be used.  Opens     */
             /* response file, if appropriate.                           */

             scalar change-in-output; /* flag: TRUE - prompt user for    */
                 /* how to change output.                                */

             Prompt for change-in-output;
             Read change-in-output;
             if change-in-output
                 then begin
                     Prompt for output form: CRT, file, both CRT&file;
                     Read output form;
                     Set CRT-output-flag;
                     Set file-output-flag;
                     if file-output-flag
                         then
                             Prompt for response-file name;
                             Read response-file-name;
                             Open response-file-name;
                     end if;
             end if;
         end procedure;
```

```
procedure RESPONSE( input: response );
    /* Outputs the response.                                          */

record response; /* to be output */

    if CRT-output-flag
        then
            Print response on CRT;
    end if;
    if file-output-flag
        then
            Write response in file( response-file-name );
    end if;
end procedure;
```

```
procedure FINISH;
    /* Carries out whatever processing is needed when a subsession */
    /* is completed, closes response-file-name if appropriate.     */

    if file-output-flag
        then
            close( response-file-name );
    end if;
end procedure;
```

INITIAL DISTRIBUTION LIST

Defense Technical Information Center 2
Cameron Station
Alexandria, VA 22314

Dudley Knox Library 2
Code 0142
Naval Postgraduate School
Monterey, CA 93940

Office of Research Administration 1
Code 012A
Naval Postgraduate School
Monterey, CA 93940

Chairman, Code 52Hq 100
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93940