

1986-06

# Implementation of the primary operation, retrieve-common, of the multi-backend database system (MBDS).

Hunt, Andrew L.

# NAVAL POSTGRADUATE SCHOOL
## Monterey, California


# THESIS

IMPLEMENTATION OF THE PRIMARY OPERATION,
RETRIEVE-COMMON, OF THE
MULTI-BACKEND DATABASE SYSTEM (MBDS)

by

Andrew L. Hunt

June 1986

Thesis Advisor: David K. Hsiao

Approved for public release; distribution is unlimited.

ECURITY CLASSIFICATION OF THIS PAGE

# REPORT DOCUMENTATION PAGE

| | |
|---|---|
| a REPORT SECURITY CLASSIFICATION<br>UNCLASSIFIED | 1b. RESTRICTIVE MARKINGS |
| a SECURITY CLASSIFICATION AUTHORITY | 3 DISTRIBUTION / AVAILABILITY OF REPORT<br>Approved for public release; distribution is unlimited |
| b DECLASSIFICATION / DOWNGRADING SCHEDULE | |
| PERFORMING ORGANIZATION REPORT NUMBER(S) | 5 MONITORING ORGANIZATION REPORT NUMBER(S) |
| a NAME OF PERFORMING ORGANIZATION<br>Naval Postgraduate School<br>6b OFFICE SYMBOL (If applicable)<br>52 | 7a NAME OF MONITORING ORGANIZATION<br>Naval Postgraduate School |
| c ADDRESS (City, State, and ZIP Code)<br>Monterey, CA 93943-5000 | 7b ADDRESS (City, State, and ZIP Code)<br>Monterey, CA 93943-5000 |
| a NAME OF FUNDING / SPONSORING ORGANIZATION<br>8b OFFICE SYMBOL (If applicable) | 9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER |
| c ADDRESS (City, State, and ZIP Code) | 10 SOURCE OF FUNDING NUMBERS<br>PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO |

TITLE (Include Security Classification)
Implementation of the Primary Operation, Retrieve-Common, of the Multi-Backend Database System (MBDS) UNCLASSIFIED

PERSONAL AUTHOR(S)
Andrew L. Hunt

| a TYPE OF REPORT | 13b TIME COVERED | 14 DATE OF REPORT (Year, Month, Day) | 15 PAGE COUNT |
|---|---|---|---|
| Masters Thesis | FROM _____ TO _____ | 1986 June 20 | 65 |

SUPPLEMENTARY NOTATION

| COSATI CODES | | |
|---|---|---|
| FIELD | GROUP | SUB-GROUP |
| | | |
| | | |

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Multi-backend Database System (MBDS), Attribute-based Data Model, Database Directory Structures, (Continued)

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

The multi-backend database system (MBDS) is a research effort conducted jointly by the Naval Postgraduate School and Ohio State University with the sponsorship of the STARS foundation. The MBDS is designed to overcome the capacity growth and performance gain problems of the traditional database systems and the single-backend database systems. The original MBDS supported four primary operations - INSERT, RETRIEVE, DELETE, and UPDATE. This thesis presents the implementation of a fifth primary operation, RETRIEVE-COMMON. This operation is used to merge the records of two files which satisfy a particular query and share a common value for given attributes. The preliminary design is discussed in the Naval Postgraduate Thesis "Design, Analysis and Implementation of the Primary Operation, Retrieve-Common, of the Multi-Backend Database System (MBDS)" by Hsiang-Lung Tung. (Continued)

| | |
|---|---|
| D STRIBUTION / AVAILABILITY OF ABSTRACT<br>☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS | 21 ABSTRACT SECURITY CLASSIFICATION<br>UNCLASSIFIED |
| a NAME OF RESPONSIBLE INDIVIDUAL<br>Prof. David K. Hsiao | 22b TELEPHONE (Include Area Code)<br>408-646-2168<br>22c OFFICE SYMBOL<br>52Hq |

D FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

BLOCK # 18 (Continued)

Data Manipulation Operators, Database Systems Testing and Database Computer Networks.

# Implementation of the Primary Operation, Retrieve-Common, of the Multi-Backend Database System (MBDS)

by

**Andrew L. Hunt**
Captain, United States Army
B.S., Hofstra University, 1978

Submitted in partial fulfillment of the
requirements for the degree of

**MASTER OF SCIENCE IN COMPUTER SCIENCE**

from the

NAVAL POSTGRADUATE SCHOOL
June 1986

# ABSTRACT

The multi-backend database system(MBDS) is a research effort conducted jointly by the Naval Postgraduate School and Ohio State University with the sponsorship of the STARS foundation. The MBDS is designed to overcome the capacity growth and performance gain problems of the traditional database systems and the single-backend database systems.

The original MBDS supported four primary operations - INSERT, RETRIEVE, DELETE, and UPDATE. This thesis presents the implementation of a fifth primary operation. RETRIEVE-COMMON. This operation is used to merge the records of two files which satisfy a particular query and share a common value for given attributes. The preliminary design is discussed in the Naval Postgraduate Thesis "Design, Analysis and Implementation of the Primary Operation. Retrieve-Common. of the Multi-Backend Database System (MBDS)" by Hsiang-Lung Tung.

TABLE OF CONTENTS

# ACKNOWLEDGEMENT

This thesis is part of ongoing database systems research efforts being conducted at the Laboratory for Database Systems Research at the Naval Postgraduate School, Monterey, Ca 93943, under the direction of Dr. David K. Hsiao. This research is supported by grants from the Department of Defense STARS Program, and from the Office of Naval Research.

I would like to express my sincere appreciation to the following individuals:

Dr. David K. Hsiao for his guidance and wisdom in leading me through this effort.

Steve Demurjian, a PhD student in Computer Science, who is currently serving as a research assistant at the Naval Postgraduate School. He provided technical support during all phases of this thesis and proved to be an invaluable asset to the MBDS project.

Finally, a special thanks goes to my wife, Debby, for her typing support and, more importantly, her patience and understanding during the past nine months.

# I. AN INTRODUCTION

Database computers must be capable of performing operations on data in a timely and efficient manner. Applications require data to be stored on auxiliary storage devices and only through an efficient organization of software and hardware can the database operations be performed quickly. The **multi-backend database system** (MBDS) at the Laboratory for Database Systems Research at the Naval Postgraduate School in Monterey, California is one such database computer. [Ref. 1].

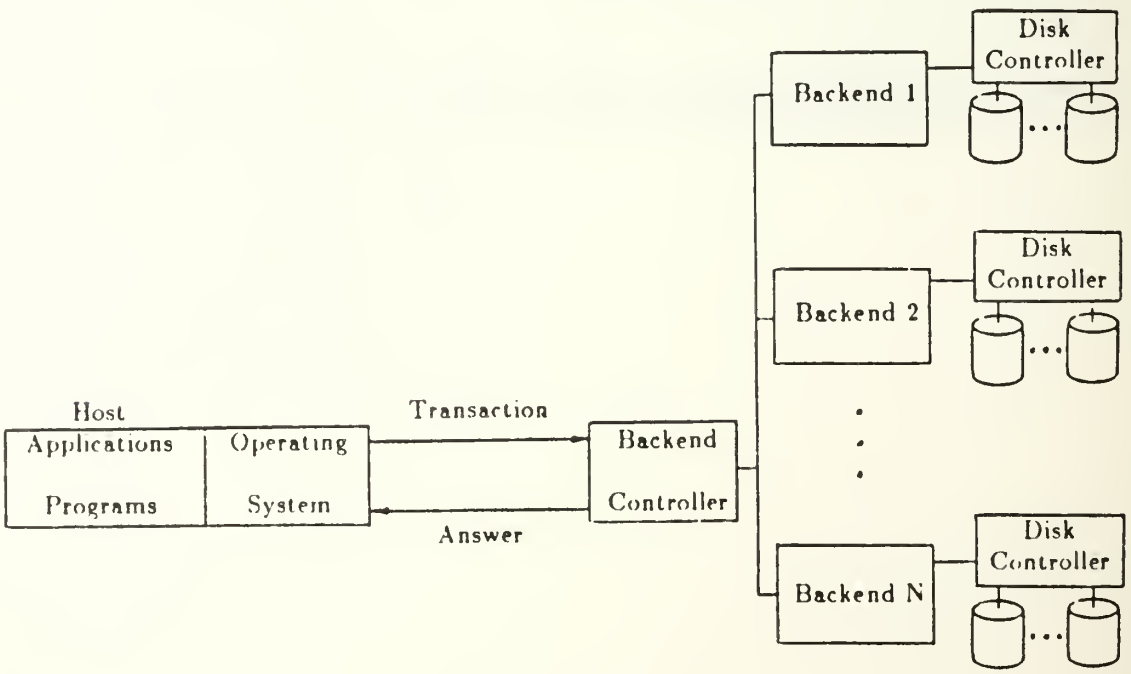


Figure 1.1 The Multi-Backend Database System (MBDS)

MBDS consists of one **controller** and one or more **backend** computers. As can be seen in Figure 1.1, requests to access the database are entered in MBDS from either a host computer or a terminal. The controller receives the requests, checks their validity, and broadcasts the requests to all of the backends. The backends are independent computers and each of them has a dedicated disk system. The database records are evenly distributed across the backends with each record being stored in only one backend's disk drive. A backend can begin processing a request the moment it is received. This allows parallel processing of the request. Since the requests can also be queued at the backend allowing for greater utilization of resources, it allows concurrent processing of requests. The goal is for MBDS to provide results in a time proportional to the number of backends connected. Thus, two backends should perform nearly twice as fast as a single backend when the database size remains constant. While the backends are performing the database operations, the controller is available for more requests to be accepted and for providing the output. This design allows the continuous processing of requests with no bottlenecks.

The controller and backends communicate using an Ethernet. The data placed on this bus is in a message format which includes the sender of the message, the receiver of the message, the type of message, and if appropriate, a message body. Each computer connected to the Ethernet has routines which allow the computer to place messages on the bus via the Put-NET process and extract messages from the bus via the Get-NET process.

Originally, MBDS had four primary operations - INSERT, DELETE, UPDATE, and RETRIEVE. This thesis presents the implementation of the fifth primary operation, RETRIEVE-COMMON. The design and analysis of this operation is performed by Hsiang-Lung Tung in his NPS Thesis "Design, Analysis, and Implementation of the Primary Operation, Retrieve-Common, of the Multi-Backend Database System". The retrieve-common operation, similar to a relational join, allows the records of two files to be combined if the records share a common value for a prescribed category. This is a powerful and desirable function with many useful applications. As an example, suppose a motor vehicle department maintains two files, namely, one for licensed drivers and the other for registered vehicles. In order to store as little redundant data as possible, the only field which these two files have in common is the social security number of the driver of one file and owner of the other file. Using the retrieve-common, it is possible to merge these two files on the common values of the social security number. The result will be longer records with all the personnel data of the licensed operator connected with the vehicle data of the vehicle registered to the same social security number. Further, from this information it is trivial to trace a license plate number to the owner of a vehicle and his address.

The rest of this chapter presents background information to familiarize the reader with MBDS. A brief discussion of the design principles which MBDS is based on is presented in the next section. The organization of the software is presented and the relationship between the existing processes and this thesis is

discussed. Finally, we present an overview of the remaining chapters in this thesis.

## A. THE DESIGN REQUIREMENTS

Three requirements have been defined for the design of MBDS. First, MBDS must be easily expandable. There are two reasons for expanding a database system. First, the existing hardware may not be large enough to hold the volume of records needed in the database. Second, the existing system is slow in providing results. These two reasons are commonly referred to as the **capacity-growth problem** and the **performance-gain problem**, respectively. MBDS assures expandability by providing identical software and hardware to all backends. The controller needs to know the number of backends being used and this is provided by the user during the initial start-up.

The second design requirement is that the hardware and software be generic. This allows for a system which can be easily expanded without regard to special hardware features. The software must be portable so that when a new backend is added the code can be transferred from an existing backend without regard to the make or model. The result of this design requirement is that MBDS can be upgraded by adding the state-of-the-art equipment to an existing system.

The third requirement suggests that the parallelism is to be exploited. This requires that the records are to be evenly distributed across the backends. Over the time, this allows the workload of each backend to be the same. Further,

requests are processed concurrently, allowing parallel accesses to the database. It is even possible for backends to be processing different requests at any instant. Since each backend can queue requests, once a backend has completed one request it can check the queue to see if further requests have been queued for processing.

These design requirements have resulted in an efficient, highly-utilized, portable system which may overcome the performance-gain and capacity-growth problems.

## B. THE PROCESS STRUCTURE

The software of MBDS is organized into a process structure as seen in Figure 1.2. The host computer has one process, the **test interface** (TI). TI provides the user with three levels of menus. Level 1 is used to provide system commands. Level 2 provides database initialization commands such as generating a database, loading a database, executing the request interface, and exiting from the system. The third menu is invoked when the option for executing the request interface has been chosen. This menu allows the user to select a particular file of requests to be executed, modify an existing file of requests, store the output received from requests to a particular file, and save the results of execution-related tasks. The test interface also provides for the format of output upon completion of a request.

### 1. The Processes of the Controller

The controller has five processes. The **Request Preparation** (REQP) process receives requests from the test interface and formats the request. If the

request is syntactically correct. REQP sends the newly formated request to the backends. The **Insert Information Generation** (IIG) process is used to

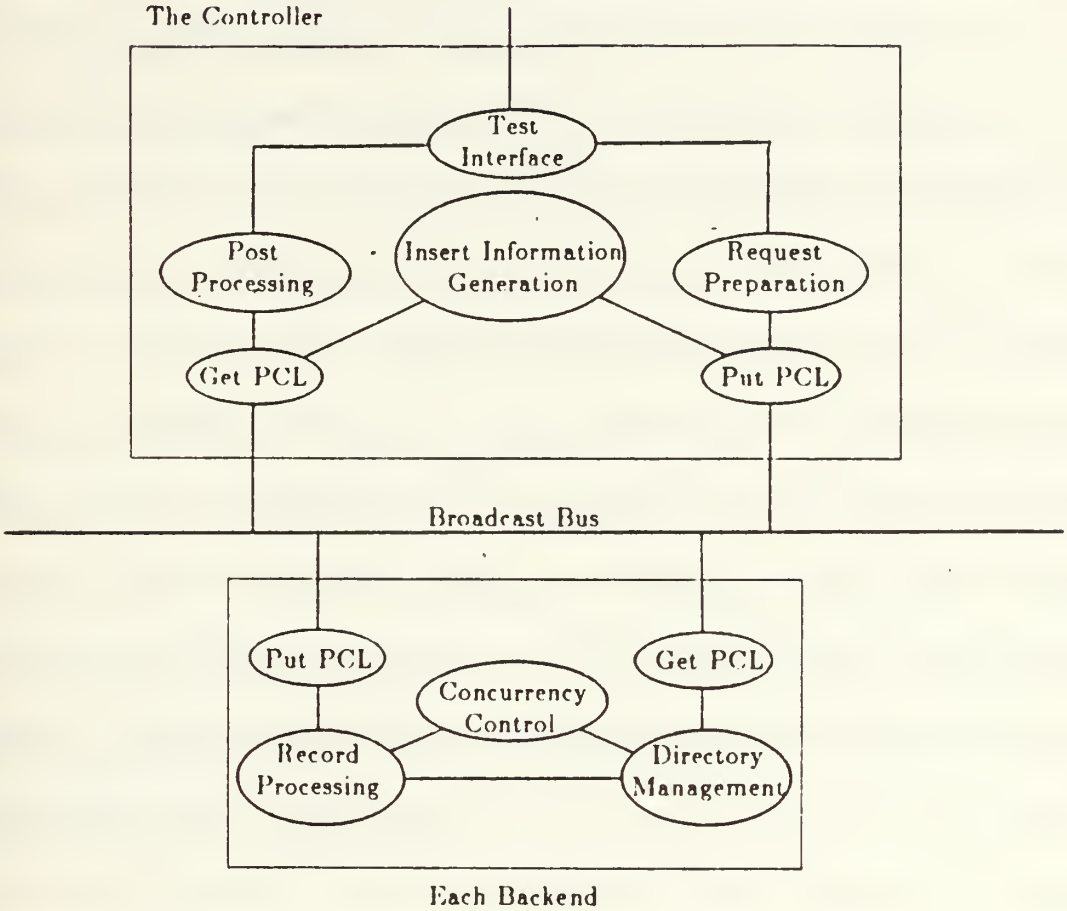


Figure 1.2 The MBDS Process Structure

provide information to the backends for insert requests. Inserts must be handled specially to insure that the data is distributed evenly across the backends. The **Post Processing** (PP) process is used to terminate a request and provide any results which may have been received from the backends to the user. **Get-NET** (GNET) and **Put-NET** (PNET) are the processes responsible for sending and

receiving messages on Ethernet. These processes determine the receiver of the message and the route of the message to the receiving process.

2. The Processes of the Backends

In addition to the two communications processes, GNET and PNET, each backend has three processes. The **Directory Management** (DM) process is responsible for identifying the secondary storage addresses necessary to access records. To perform this task, three tables are maintained - the **Attribute Table** (AT), the **Descriptor-to-Descriptor-Id Table** (DDIT), and the **Cluster-Definition Table** (CDT). AT maps directory attributes to their descriptors; DDIT assigns a unique id to each descriptor; and CDT maps descriptor ids to cluster ids. By referencing these tables, DM is able to determine disk addresses of those records in which results of a particular request may be found. This method, referred to as **clustering**, provides for an efficient technique to access only the relevant portion of the secondary storage. The **Concurrency Control** (CC) process is used to guarantee consistency of directory data and user data. MBDS allows descriptors, clusters, and secondary storage addresses to be changed dynamically. Thus, CC must restrict and control their access during such changes. The **Record Processing** (RECP) process receives the request and disk addresses from DM, accesses the secondary storage for the addressed records, and executes the request against the records. It is in this process that the retrieve-common operation is implemented.

## C. THE ORGANIZATION OF THE THESIS

The remainder of the thesis is organized as follows. In Chapter II we present the four original primary operations and provide the syntax and an example of the new primary operation, retrieve-common. In Chapter III we present a detailed examination of the processing logic of the new operation. This is followed by necessary changes to the original specifications as they are required in the implementation. Chapter IV provides the data structures which have been added or modified in the course of the implementation of the operation. A walk-through the user interface is presented in Chapter V to provide users with the details of how to execute a retrieve-common request. In Chapter VI the test procedures used to verify the program's correctness is discussed. Finally. Chapter VII concludes this thesis. This thesis is intended to present the details of the implementation of the retrieve-common and provide the guidance necessary to properly use the new operation.

## II. THE PRIMARY OPERATIONS OF MBDS

In this chapter we present the primary database operations of MBDS. We first present the attribute-based data model of MBDS to orient the reader to the terminology used with the operations. Then, we discuss the four original operations and provide examples of each. Finally we present the syntax and an example of the retrieve-common operation. This chapter concludes with an overview of the processing for the retrieve-common operation.

### A. THE ATTRIBUTE-BASED DATA MODEL

MBDS uses an attribute-based model in the design of the database [Ref. 1 : pp 9-14]. This model calls for records to be stored as a set of attribute-value pairs. The **attribute** s used to describe a class or certain characteristic of which the values are a part. The second component is a value for the attribute. This value can be a string or an integer, but all values for a given attribute must be consistent. Attribute-value pairs are enclosed in brackets such as:

<STATE, Vt>

A **record** is a grouping of attribute-value pairs and a record body where no attribute appears more than once in the record. The record body consists of a string of textual data. We use the brackets, {, }, to enclose the record body. A **file** is used to name a collection of records which are grouped under common

characteristics and formats. say. having a similar set of attributes. Finally. several files of related data are together referred to as a **database**.

## B. THE DIRECTORY STRUCTURES

MBDS uses directory data to manage the database. Three constructs are used for this function - attributes, descriptors, and clusters. As described above, attributes describe the category or characteristic of the user data. **Descriptors** provide ranges into which the attribute values may be partitioned. For example, the attribute "U.S. States" could use a set of three descriptors [Alabama - Hawaii], [Idaho - Texas], and [Utah - Wyoming]. Observe that any set of descriptors must be mutually exclusive. From the user defined descriptors, clusters are formed. A **cluster** is a group of records such that every record in the cluster satisfies the same descriptor. For instance, using the above set of descriptors, records containing the U.S. States such as Alaska. Connecticut. and Delaware must be in the same cluster because their U.S. State attribute values alphabetically fall between Alabama and Hawaii. Clustering is a very important principle for MBDS because it allows indexing and thus only those clusters which may have data which satisfies a request will be accessed during processing.

## C. THE DATA MANIPULATION OPERATIONS

The original MBDS recognized four primary operations to support the database. These four are INSERT, DELETE. UPDATE, and RETRIEVE.

The INSERT operation is used to place a record into the database. The syntax for this request is:

INSERT Record

where Record is a set of attribute-value pairs and record body. The following example inserts a record into the US file:

INSERT (<FILE, US>, <STATE, Vt>, <TOWN, Morgan>, {...})

The DELETE operation is used to remove a record or set of records from the database. The syntax for this request is:

DELETE Query

The query (in parenthesis) is used to select the records which the operation will be performed on. For instance, the following example deletes all Vermont records from the US file:

DELETE ((FILE = US) and (STATE = Vt))

The UPDATE operation is used to modify records which are already in the database. The syntax for the UPDATE request is:

UPDATE Query [Modifier]

The query is used to select the records which will be updated and the modifier (in brackets) specifies the type of change which will be performed on those records. The following example will change the population of Morgan, Vt to 500:

UPDATE ((FILE = US) and (STATE = Vt) and (TOWN = Morgan)) <POP = 500>

The RETRIEVE operation is used to locate and return to the user those records which satisfy the query. The syntax of a retrieve request is:

RETRIEVE Query (target-list)[BY Attribute][WITH Pointer]

The query specifies which records are to be selected. The target-list provides the attribute values of that record which are to be returned to the user. The BY clause, as an option, is used with the aggregate operations. AVG. COUNT. SUM, MIN. and MAX. When this clause is used, the records selected are grouped to perform the aggregate operation. The WITH clause, as an option, is used to return pointers to the retrieved records for later use with an UPDATE request. An example of a retrieve which will return all states in the United States with a town named Morgan is:

RETRIEVE ((FILE = US) and (TOWN = Morgan)) (STATE)

In this example the query is (FILE = US) and (TOWN = Morgan). The target-list is STATE. The BY and WITH clauses are not used.

## D. THE RETRIEVE-COMMON OPERATION

### 1. The Syntax of the Retrieve-Common Operation

The retrieve-common operation is used to merge the records of two files which share a common value for specified attributes. The syntax for the retrieve-common resembles the syntax of the retrieve request. This allows the actual

selection of records from secondary storage to proceed exactly as two retrieve requests. The syntax for the retrieve-common is:

```
RETRIEVE Query-1 (target-list-1)[BY Attribute][WITH Pointer]
COMMON (Attribute-1, Attribute-2)
RETRIEVE Query-2 (target-list-2)[BY Attribute][WITH Pointer]
```

The first retrieve is referred to as the **source retrieve** and the second retrieve is called the **target retrieve**. **Attribute-1** references those records which satisfy the source retrieve and **Attribute-2** is used with the target retrieve. These attributes are used to select the values which must be identical in order to connect a source record to a target record. These two attributes need not be the same for the merge to occur, only their values do. As an example:

```
RETRIEVE (FILE = US) (STATE, TOWN)
COMMON (TOWN, CITY)
RETRIEVE (FILE = Canada)(PROV, CITY)
```

could return:

(<STATE, Vt><TOWN, Morgan><PROV, Quebec><CITY, Morgan>)

or more generally, for the United States and Canada find all states, towns, provinces, and cities, where the town and city are identical.

2. An Overview of the Processing

The logical operation of the retrieve-common is as follows:

a. The retrieve-common request is modified into two retrieve requests by placing the common attributes into the target list of the source retrieve and the target retrieve, respectively.

b. All of the records which satisfy the source retrieve are gathered, the common attribute value is hashed, the records are placed in the virtual memory, and the hashed addresses are stored in the hash tables.

c. All of the records which satisfy the target retrieve are collected, and the hash values are calculated. These records are also placed in the virtual memory and their addresses are stored into another hash table.

d. The target records of the backend are transmitted to all of the other backends to be hashed with the local target records. In this way, each backend has only local source records, but has every target record which is in the database.

e. To perform the pairwise merge, the backend checks if the first value in each source record is the same as the first value in each target record, since the common attributes have been placed in the front of the target list so that their values would be the first value listed for all records. If the two values are the same, the records are concatenated and outputted.

# III. MODIFICATIONS REQUIRED TO SPECIFICATIONS

The original specifications for the retrieve-common operation are presented in [Ref 2]. When reviewing the specifications for the initial stages of the implementation we have found a number of basic errors and oversights. These basic errors have been quickly and easily corrected. Throughout the course of the -mplementation and integration we have also encountered some major logical and program flow errors. These major errors resulted in a partial redesign of the program flow regarding the execution of the retrieve-common request. In the rest of this chapter. we first present the execution steps for the retrieve-common operation. Then we detail all of the modifications to the original specifications.

## A. THE EXECUTION OF THE RETRIEVE-COMMON REQUEST

This section describes the sequence of actions for executing a retrieve-common request. Figure 3.1 lists all of the types of messages used to control MBDS. Figure 3.2 displays the controller and backend processes and the messages which are passed within the system for a retrieve-common operation. The order in which the messages are passed is denoted alphabetically (e.g., 'a' is first). The digit following the letter is the type of message as listed in figure 3.1.

A retrieve-common request originates in the test-interface (TI) process. This process allows the user to choose whether a single request is being generated or

| MESSAGE-TYPE NUMBER AND NAME |
|---|
| 1 Traffic Unit |
| 2 Request Results |
| 3 Number of Requests in a Transaction |
| 4 Aggregate Operators |
| 5 Requests with Errors |
| 6 Parsed Traffic Unit |
| 7 New Descriptor Id |
| 8 Backend Number |
| 9 Cluster Id |
| 10 Request for New Descriptor Id |
| 11 Backend Results for a Request |
| 12 Backend Aggregate Operator Results |
| 13 Record that has Changed Clusters |
| 14 Results of a Retrieve Caused by Update |
| 15 Descriptor Ids |
| 16 Request and Disk Addresses |
| 17 Changed Cluster Response |
| 18 Fetch |
| 19 Old and New Values of Attribute being Modified |
| 20 Type-C Attributes for a Traffic Unit |
| 21 Desc-Id Groups for a Traffic Unit |
| 22 Cluster Ids for a Traffic Unit |
| 23 Release Attribute |
| 24 Release all Attributes for an Insert |
| 25 Release Descriptor-Id Groups |
| 26 Attribute Locked |
| 27 Descriptor-Id Groups Locked |
| 28 Cluster Ids Locked |
| 29 Generated Inserts Completed |
| 30 Request Id of a Completed Request |
| 31 Update Request has Completed |
| 32 Source Retrieve-Common has Completed |
| 33 Notification of a Retrieve-Common Request |
| 34 Target Retrieve-Common Records |

Figure 3.1 The MBDS Message Types

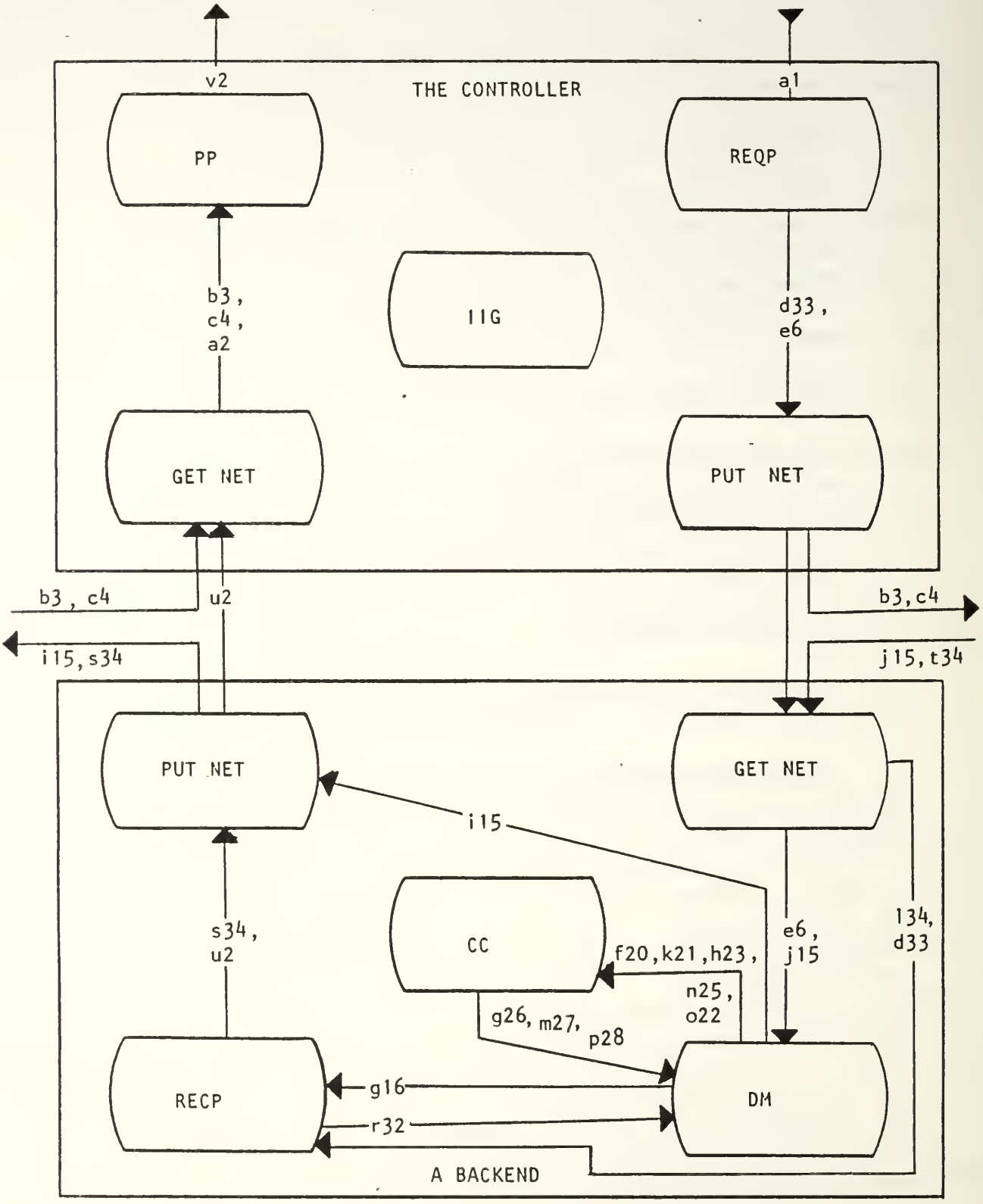


FIGURE 3.2

The Sequence of Messages for
Executing a Retrieve-Common Request

whether a transaction of several requests is being built. As each request is composed, the user only inputs that information which changes from one request to another such as the query and the target list. The user is not responsible for the format of the request; this is automatically generated.

The request is then sent to the Request Preparation (REQP) process for parsing, syntax checking, and formatting into a request table (a1). REQP notifies Post Processing (PP) of the number of requests in the transaction (b3) and the aggregate operator of the request (c4). Upon completion of these actions, REQP notifies Record Processing (RECP) of the new retrieve-common request (d33). REQP then sends the parsed traffic unit to Directory Management (DM) (e6). DM calls on the Concurrency Control (CC) process to lock the directory attributes (f20). After they have been locked, CC notifies DM (g26) and DM begins descriptor search for both the source and target retrieve. Once this is completed, DM notifies CC to release the locks on the attributes (h23) and DM broadcasts the descriptor ids of the source and target retrieves to the other backends (i15). The DM in the other backends are also sending their descriptor-ids to the DM in this backend (j15). The backends use the information received from the other backends to form descriptor-id groups. These groups are then sent to CC to be locked (k21). After CC notifies DM that these groups are locked, (m27) DM performs cluster search and notifies CC to release the locks for both retrieves (n25). Next, DM sends the cluster ids for the retrieval to CC (o22). CC notifies DM when the clusters have been locked (p28). At this time, DM

determines the disk addresses for this request. DM then sends the source retrieve request and its disk addresses to RECP (q16). When RECP finishes executing the source retrieve and has stored the records in the virtual memory, it notifies DM that the source retrieve has completed processing and the target retrieve and its disk addresses can be sent(r32). RECP performs the necessary processing for the target retrieve and stores the records which satisfy the request into the virtual memory. The last record stored for this request carries a completion flag which indicates that the last target record has been stored. This flag signals the backend to broadcast the target records to the other backends (s34). The backends must closely monitor the progress of the two requests because target records from other backends may be received (t34) before the source retrieve has been processed at this backend. Two flags are used to control this - a source completed flag, and a target count flag which counts the number of backends having finished sending target records. The pairwise merge of records can not be started until both the source retrieve has completed and all of the records from the other backends have been received. Once the merging of records has been completed, the results are sent to PP (u2) which removes the request from the active request table and sends the records to TI (v2) for formatting the output and sending the results to the host.

## B. THE DEVIATIONS FROM THE SPECIFICATIONS

Since the time that the specifications have been designed [Tun85], several versions of MBDS have been implemented. These versions have been combined, resulting in a system which uses multi-templates, multi-computers, and the retrieve-common operation. Differences in the specifications are not discussed. Interested readers should consult [Ref 3] and [Ref 4] for more information. Presented in the following paragraphs are errors and omissions of the specifications listed by process. Two procedures, Insert Information Generation (IIG) and PP have not required any modification.

### 1. The Test Interface Process

The TI process has been omitted from the retrieve-common specifications. This process performs the vital functions of receiving the input from the host computer and displaying the results. A new procedure, TI-retrieve-common, has been implemented to allow the user to build a new retrieve-common request. This procedure makes use of the existing "build-retrieve" procedure by calling it twice, once to build the source retrieve and once to build the target retrieve. Placed between these two calls is a procedure which builds the common attributes and places them in the proper format. The procedure TI-ReqRes-Output has been modified to provide a format compatible with the attribute-based model, that is, records formed as attribute-value pairs.

2. The Request Preparation Process

The specifications of the REQP process accurately described the method to properly parse the retrieve-common request. One deviation from the pseudo-code which the implementation used has been for style more than accuracy. The specifications called for five flags to be used to recognize which attribute in the retrieve-common is being parsed. For instance, if flag1 is true, the common attribute-1 is being parsed, if flag2 is true, then common-attribute-2 is being parsed, etc. These flags resulted in a long if-then-else ladder. Rather than using these flags, a single variable is used which ranges in value from zero to five. A case statement is used to determine which attribute is being parsed based on the value of the variable.

Another modification, which has not been anticipated in the specifications, is the need to notify RECP of a retrieve-common request prior to notifying DM. During testing, it has become apparent that some backends broadcast target records before the other backends are aware that the request exists. This results in the backend not having a buffer available to store the incoming records. The key to solving this problem is to have REQP notify RECP of the request number of the retrieve-common. Then, RECP immediately creates buffer space for both the source-retrieve and the target-retrieve.

3. The Concurrency Control Process

There has been a need to add the retrieve-common type to case statements in some of the procedures, but the logical flow remained the same as for the retrieve request.

4. The Parallel Communications Link Process

The modifications which were required to the PCL processes included adding the three new message types (SourceFinished, BucketInformation, and RP-RetComNotification) with the routing instructions for each message and specifying which process is to receive messages of each type.

5. The Directory Management Process

The original specifications called for DM to hold the target-retrieve from all processing until the source-retrieve had been completed. Rather than hold the target-retrieve processing at this point, both retrieves perform attribute search, descriptor search, cluster search, and address generation concurrently. After address generation, while the source retrieve is sent to RECP, the target retrieve is held at DM. This requires a buffer to store the disk addresses for the target retrieve. When notification arrives that the source-retrieve has finished, the request and disk addresses are sent to RECP for processing.

6. The Record Processing Process

The majority of the retrieve-common implementation occurs in this process. Almost 1200 lines of source code have been added. Due to the

complexity of the message passing and processing logic, several differences exist between the specifications and the implementation.

As discussed in the REQP section above, a message is sent from REQP to RECP announcing a new retrieve-common request is being processed. This message activates a routine which allocates and partially initializes a RP-rid-info structure. This structure is used by RECP to store all of the information known about the request. This initialization includes allocating hash tables in the event that target records arrive from other backends which may be processing this request quicker. The structure can only be partially initialized at this time because DM is using the request table and generating the disk addresses which will be needed to complete this structure. When the disk addresses have been generated, DM sends this information to RECPROC and the rest of the initialization takes place.

The specifications call for a global table to be available which provides the disk addresses of the hash tables. Rather than using a global structure, a hashing-information structure is used which is attached to the RP-rid-info structure. This structure is discussed in detail in Chapter IV.

The hashing procedures have been implemented as written in the specifications; however, rather than three lengthy procedures for small integers, large integers, and strings, one procedure is used. The only difference in the way these three types of values are processed is the manner in which the hash value (bucket) is calculated. For this part of the logic a case statement is used.

The final change to the specifications has been the method of managing auxiliary memory. The specifications require a deliberate step to be made by placing the hash tables into known disk addresses. Rather than performing this task, the implementation uses the logical addresses for the hash tables and lets the operating system page the records into and out of the secondary storage. This greatly reduces the complexity of the code.

## IV. THE DESIGN OF DATA STRUCTURES FOR THE IMPLEMENTATION

The data structures in the implementation of the retrieve-common operation are selected based on three design requirements. First, the data structures must provide a buffer for the selected source records and target records to be stored. Second, the retrieve-common data structures should be hidden from data structures used by other operations. This results in reducing the possibility of other operations inadvertently using the retrieve-common structures. The third requirement is for the retrieve-common structures to use the memory efficiently. To meet this third requirement, the data structures should be dynamically allocated only when a retrieve-common operation is processed. Other operations should not be allocated any of the retrieve-common structures.

To satisfy these requirements, only the record processing data structures are modified. The following sections describe the modifications to the existing structures and the creation of new structures from a top-down viewpoint.

### A. THE RP_RID_INFO MODIFICATION

The RP_rid_info (Record Processing request identification information) structure is a high-level structure which is allocated each time RECP is notified of a new request. This structure holds information about the request such as a pointer to the request number, a pointer to the request table, a pointer to the

result buffer. and a pointer to the new request which has been received. as can be seen in Figure 4.1.

```
struct RP_rid_info
    {
    struct ReqId RP_ri_rid;
    struct REQtbl_definition RP_ri_req;
    struct rtemp_definition *RP_ri_tmpl_ptr;
            .
            .
            .
    struct hashing_info *RP_ri_hash;
    int    SrceDone;
            .
            .
            .
    struct ResultBuffer *RB_pointer; /* result buffer */
    struct RP_rid_info *next_RP_rid_info;
    };
```

Figure 4.1 The RP_Rid_Info Structure

This structure is used to link the retrieve-common structures from the existing data structures. The hash tables are not placed directly into this structure, but are accessed by a pointer in this structure called RP_ri_hash. This allows a better use of memory because non-retrieve-common requests set this pointer to NULL and no further retrieve-common structures are allocated. Retrieve-common requests use this pointer to connect a hashing-info structure which is discussed in the next section. The second modification required is to add a boolean flag

which is set to true when the source retrieve has completed. This flag is used to determine if the merging of files can begin. Recall that there are two criteria which must be satisfied before merging can occur, namely, all the target records from the other backends must be received and the source and target retrieves for this backend must be completed.

## B. THE HASHING_INFO STRUCTURE

The hashing_info structure, used to hold the record buffer and valuable control information, is essential for the efficiency of the code. Before discussing this structure, an overview of the retrieve-common processing is necessary.

For both the source and target retrieves of the retrieve-common operation, records are physically removed from the disk using the code developed for the retrieve command. Just prior to the point as in the retrieve request when the result records are passed back to the controller from the backends, the backend realizes that the current operation is the retrieve-common request, not the retrieve. This signals the backend not to send the records to the controller but to store the records in the virtual memory. To do this, each record is hashed on its common attribute value, and then stored in a temporary buffer. As this buffer becomes full, the records are transferred to the virtual memory and the virtual

address is placed in the hash tables. When the last record for a request has been placed in the virtual memory, several options may occur:

1. If this is a target-retrieve, the target records which are accessible from the hash tables are sent to the other backends.

2. If all of the target records from the other backends have been received and the source and target retrieves at this backend have been completed, the pairwise merge can begin.

3. If either its local target retrieve has not been completed or the target records of at least one other backend have not been received, this backend must wait before beginning to merge the records.

The retrieve-common is allocated with two RP_rid_info structures when the record processing is notified of the request, one for the source retrieve and one for the target retrieve. Certain initializations must occur at this time including the allocation of the hashing_info structure. When the first record to be stored is received, further initialization occurs. The value of the common attribute is used to determine the type of hashing functions to be used for this retrieve. It can be one of three types - string, small integer, or large integer. To make this determination, the template information must be examined. The template holds the value type of the attribute as well as the maximum and minimum values that this attribute can assume. Once this information is gathered from the template, it is not collected again because all the records of this request use the same template and share the same characteristics. For this reason, the hashing_info structure

holds the value type, minimum value, and the range of values. A new_request_flag is used to determine whether this initial examination of the template has occurred. The target_counter is used to count the number of backends which have sent target records for this request. This flag is incremented each time another backend informs this backend that the last record has been sent.

```
struct hashing_info
    {
    struct hash_result hash_buffer;
    int value_type;
    int min,
        range,
        new_request_flag,
        target_cnt;
    struct block *hash_table[NUMBER_OF_BUCKETS];
    };
```

Figure 4.2 The Hashing_Info Structure

After the hash value has been calculated for each record, the record is not immediately placed in the hash table but is placed in a temporary buffer called the hash_buffer. The process of placing records in the hash tables can be lengthy and for this reason the records are first buffered and then as the buffer gets full, several records can be placed in the hash tables at one time.

The final element in the hashing_info structure is the array of hash tables. The hash tables are pointers to the block structure which stores the physical records. These pointers are organized into an array with the index of the array being the hash value. Using this convention, the records which receive a hash value 223 will be found by referencing hash_table[223]. The composition of the block structure is discussed in section D.

## C. THE HASH_RESULT STRUCTURE

The purpose of the hash_result structure shown in Figure 4.3 is to provide a temporary buffer for records which are not yet stored in the hash tables.

```
struct  hash_result
    {
    struct RP_rid_info *Origin_RP_ri_ptr;
    int length;
    char hashed_result[HR_SIZE + 1];
    };
```

Figure 4.3 The Hash_Result Structure

The primary element of the hash_result is the character array which is used to store the hashed records, called the hashed_result. The index for this array is the

variable length which is also an element of this structure. The third element of this structure is the Origin_RP_ri_ptr. This pointer links back to the current RP_ri_ptr and is used so that the high-level information such as the request identification number and the request table can be accessed from the lower levels.

## D. THE BLOCK STRUCTURE

The block structure shown in Figure 4.4 holds the records for each hash value. The element of the structure which holds the records is the

```
struct block
   {
    int length:
    struct block *next_block:
    char contents[MAX_BLK_SIZE];
   };
```

Figure 4.4 The Block Structure

character element of the structure which holds the records is the character array, called contents. The index for this array is the variable length. The third element of this structure is a pointer to the next block, called next_block. The number and size of records which must be stored under the index of one hash value is virtually limitless. Dynamic allocation of blocks must occur when records cause an overflow to the last block. This pointer is used to connect the blocks of one hash table into a linked list.

# V. THE TESTING

The testing of a software project is the single most important task in the software life cycle. It is during this stage of the software development that the requirements definition, detailed design, and implementation are evaluated and any errors which are discovered are corrected. The objective of the testing phase is to locate as many errors as possible. It is very difficult to determine when a program is complete and correct. There is a well-known saying that the bug to worry about is the one which has not been found. This is certainly true for projects as large as MBDS.

In the first section of this chapter we present an overview of the types of tests which we have determined are appropriate for the retrieve-common implementation. The final section of this chapter discusses the unit tests which have been conducted.

## A. THE TESTING PROCESS

Several techniques have been used to test the retrieve-common implementation. These techniques include **unit testing**, **white-box testing**, and **black-box testing**. During a unit test, segments of code are tested in an isolated environment in an attempt to determine the code's correctness. This allows the location of errors to be discovered easier than if the complete code is

being tested. During white-box and black-box testings several retrieve-common requests are used to test an implementation. A white-box test is used to demonstrate that every line of code properly performs the desired operation. This implies that each path of control structures (e.g., loops and if-statements) is tested. A black-box test is used to demonstrate that the program is correct even for several different types of input. This includes boundary cases, that is, positive, negative, and zero input values, and inputs which are close to satisfying a request but off enough to make them wrong.

Unit testing in the strictest sense, as single modules, is not possible for the retrieve-common operation because very few of the modules can stand alone. However, we have been able to divide the program into three sections and test each of these sections individually. The first section has been tested for the modifications to the REQP_process. The second section has been verified for the correctness of the hashing algorithm in a single backend. The final section has been tested for the broadcasting and receiving of target records in a multiple backend environment. These tests are discussed in the next section of this chapter.

Although it is extremely difficult to test all combinations of the control structures in the retrieve-common implementation, the white-box test did test each processing sequence. For the retrieve-common operation there are several special cases which have been considered. These included testing for common

attributes which are of value types string, small integer, and large integer, as well as intermediate and final results which overflow the buffer spaces.

The black-box test was used to test unusual inputs. For this test the retrieve-common implementation has been tested for requests which have no records to be returned, requests which return a large number of records, and requests with nearly identical common attributes values.

## B. THE DETAILS OF THE TESTING PROCESS

In this section, we discuss the purpose of each phase of the unit testing. This is followed by the method of measuring the results. The discussion of each phase is concluded with the significant errors which have been discovered during the phase of testing and the corrective action which has been required.

The first phase tested the modifications to the REQP modules. To conduct this test, the necessary modifications to the code have been made and a retrieve-common request is inputted. Recall that the retrieve-common operation is processed the same as two retrieve requests. Since the hashing function has not been implemented at the time of the test, the results from the processing of the source and target retrieves are sent to the host. This test is considered complete when these results are received at the host computer. Two significant errors have been discovered during this test. One error had been the result of using an outdated version of the lexical source code. An updated version had been implemented just prior to the start of the retrieve-common implementation.

Once this error had been corrected, the execution had been traced from TI to the source code of the compiler. The second error had been in the format of the request when sent from REQP to DM. Under close examination of the format, we observed that the number identifying the request had been initialized to zero rather than one. This error had been significant, but easy to correct. Once these errors had been corrected, the results for the source and target retrieves had been received at the host computer.

The purpose of the second phase of the unit testing had been to test the retrieve-common implementation in a single-backend environment. This test demonstrates the correctness of the modifications to the DM process, and the implementation of the hashing algorithm, the use of the virtual storage, and the output format. Errors which had been found in this section of code were minor and easily detected and corrected. These errors had included syntax errors and infinite loops caused by searching for an incorrect delimiter.

The third phase of the retrieve-common unit testing had been designed to test the message sending and message receiving functions. To perform this test, several backends had been loaded with the database records and the transfer of target records between backends had been examined. It is during this phase of testing that we had become aware that the new message type, Retrieve-Common Notification, is required as discussed in Chapter IV. It is also in this stage that we had experienced messages being lost on PCL for a large number of backends. This problem had been isolated to the Get NET process. The messages had been

properly placed on the Ethernet but had not always been received by the Get NET process. This problem has been solved by raising the priority of the Get NET process in order to guarantee that this process is the first process to check the message queue.

# VI. CONCLUSION

In this thesis we have presented the implementation of the retrieve-common operation in the multi-backend database system. The original specifications have been closely followed with the exceptions as noted in Chapter III.

The MBDS architecture consists of a controller computer and one or more backend computers. Communication between these computers is on an Ethernet. The controller manages the backends by assigning them database requests to process. The backends perform the operations required and send the results to the controller as appropriate. The software is organized in a process structure with five controller processes, five backend processes, and a test interface process for the host computer.

MBDS uses an attribute-based data model. This model represents database records as a group of attribute-value pairs and a record body. The primary operations, INSERT, DELETE, RETRIEVE, UPDATE, and now, RETRIEVE-COMMON, manipulate the data by accessing the records. Not every record is accessed for a request because the records have been partitioned, or clustered, by the values of the directory attributes. This allows MBDS to access only those disk addresses which may have relevant records in them.

The retrieve-common request is used to merge the records of two files which share a common attribute value for specified attributes. The syntax of the retrieve-common request resembles the syntax of the retrieve request. This allows the processing logic of the retrieve-common operation to use many of the retrieve procedures. The format of a retrieve-common request is a source retrieve request followed by a common attribute for the source request, a common attribute for the target retrieve request, and a target retrieve request. Using this format, a retrieve-common request can be transformed into two retrieve requests by placing the respective common attributes into the source and target retrieves.

The processing of a retrieve-common request differs from the retrieve request after the records have been fetched from the auxiliary memory device. Rather than return the results to the host, they are hashed on the common attribute value and placed into the virtual storage with the logical address placed in a hash table. This occurs for both the source and target retrieve. The target retrieve results are then sent to all other backends so that each backend has only local source retrieve results but have all the target retrieve results in the system. After these additional records have been stored in the virtual storage of the backend, each source retrieve record is compared with each target retrieve records. If the respective common attributes are identical the two records are merged and the results are sent to the host computer.

In summary, we have implemented an extremely useful and desirable function. The retrieve-common operation makes MBDS more complete by

providing a method of merging two files. With the completion of this thesis. the final MBDS primary operation has been implemented.

## APPENDIX A

## THE TESTING RESULTS

In this appendix, we present the test results for the retrieve-common operation.

### A. THE INSERTED RECORDS

TEST 001[INSERT(<TEMP,Part>,<PNO,P1>,<NAME,Idm>,<CITY,Mont>)]
TEST 002[INSERT(<TEMP,Part>,<PNO,P2>,<NAME,Xyz>,<CITY,Sali>)]
TEST 003[INSERT(<TEMP,Part>,<PNO,P3>,<NAME,Nut>,<CITY,Colu>)]
TEST 004[INSERT(<TEMP,Sups>,<SNO,S1>,<NAME,Nut>)]
TEST 005[INSERT(<TEMP,Sups>,<SNO,S2>,<NAME,Nut>)]
TEST 006[INSERT(<TEMP,Sups>,<SNO,S2>,<NAME,Nut>)]
TEST 007[INSERT(<TEMP,Sups>,<SNO,S1>,<NAME,Dec>)]
TEST 008[INSERT(<TEMP,Sups>,<SNO,S3>,<NAME,Nut>)]
TEST 009[INSERT(<TEMP,Sups>,<SNO,S3>,<NAME,Dec>)]
TEST 010[INSERT(<TEMP,Sups>,<SNO,S4>,<NAME,Nut>)]
TEST 011[INSERT(<TEMP,Sups>,<SNO,S4>,<NAME,Dec>)]
TEST 012[INSERT(<TEMP,Ship>,<SNO,S1>,<PNO,P2>,<QTY,500>)]
TEST 013[INSERT(<TEMP,Ship>,<SNO,S2>,<PNO,P2>,<QTY,500>)]
TEST 014[INSERT(<TEMP,Ship>,<SNO,S3>,<PNO,P1>,<QTY,500>)]
TEST 015[INSERT(<TEMP,Ship>,<SNO,S4>,<PNO,P2>,<QTY,1000>)]
TEST 016[INSERT(<TEMP,Ship>,<SNO,S1>,<PNO,P2>,<QTY,1000>)]
TEST 017[INSERT(<TEMP,Ship>,<SNO,S2>,<PNO,P2>,<QTY,1000>)]
TEST 018[INSERT(<TEMP,Ship>,<SNO,S3>,<PNO,P1>,<QTY,2000>)]
TEST 019[INSERT(<TEMP,Ship>,<SNO,S4>,<PNO,P2>,<QTY,2000>)]
TEST 020[INSERT(<TEMP,Ship>,<SNO,S1>,<PNO,P2>,<QTY,2000>)]

B. TEST NUMBER 1.

The first test demonstrates that a typical retrieve is properly processed.

TEST 021 RETRIEVE(TEMP=Sups)(SNO,NAME)

SNO = S3
NAME = Nut

SNO = S4
NAME = Nut

SNO = S1
NAME = Dec

SNO = S3
NAME = Dec

SNO = S4
NAME = Dec

SNO = S1
NAME = Nut

SNO = S2
NAME = Nut

SNO = S2
NAME = Nut

C. TEST NUMBER 2.

The second test shows the results for another retrieve.

```
TEST 022[RETRIEVE(TEMP=Ship)(PNO,SNO,QTY)]

    PNO = P2
    SNO = S4
    QTY = 2000

    PNO = P2
    SNO = S1
    QTY = 2000

    PNO = P1
    SNO = S3
    QTY = 2000

    PNO = P2
    SNO = S1
    QTY = 1000

    PNO = P2
    SNO = S2
    QTY = 1000

    PNO = P1
    SNO = S3
    QTY = 500

    PNO = P2
    SNO = S1
    QTY = 500

    PNO = P2
    SNO = S2
    QTY = 500

    PNO = P2
    SNO = S4
    QTY = 1000
```

D. TEST NUMBER 3.

This test demonstrates the results of a retrieve-common request which combines the results from Test 1 and Test 2.

```
TEST 023[RETRIEVE(TEMP=Sups)(SNO,NAME)
         COMMON(SNO.SNO)
         RETRIEVE(TEMP=Ship)(PNO,SNO,QTY)]

<COMMON.File> <SNO.S1> <NAME.Dec> <PNO.P2> <SNO.S1> <QTY.2000>
<COMMON.File> <SNO.S1> <NAME.Dec> <PNO.P2> <SNO.S1> <QTY.1000>
<COMMON.File> <SNO.S1> <NAME.Nut> <PNO.P2> <SNO.S1> <QTY.500>
<COMMON.File> <SNO.S1> <NAME.Nut> <PNO.P2> <SNO.S1> <QTY.2000>
<COMMON.File> <SNO.S1> <NAME.Dec> <PNO.P2> <SNO.S1> <QTY.500>
<COMMON.File> <SNO.S3> <NAME.Nut> <PNO.P1> <SNO.S3> <QTY.2000>
<COMMON.File> <SNO.S1> <NAME.Nut> <PNO.P2> <SNO.S1> <QTY.1000>
<COMMON.File> <SNO.S2> <NAME.Nut> <PNO.P2> <SNO.S2> <QTY.500>
<COMMON.File> <SNO.S3> <NAME.Nut> <PNO.P1> <SNO.S3> <QTY.500>
<COMMON.File> <SNO.S3> <NAME.Dec> <PNO.P1> <SNO.S3> <QTY.2000>
<COMMON.File> <SNO.S2> <NAME.Nut> <PNO.P2> <SNO.S2> <QTY.1000>
<COMMON.File> <SNO.S2> <NAME.Nut> <PNO.P2> <SNO.S2> <QTY.500>
<COMMON.File> <SNO.S3> <NAME.Dec> <PNO.P1> <SNO.S3> <QTY.500>
<COMMON.File> <SNO.S4> <NAME.Nut> <PNO.P2> <SNO.S4> <QTY.2000>
<COMMON.File> <SNO.S2> <NAME.Nut> <PNO.P2> <SNO.S2> <QTY.1000>
<COMMON.File> <SNO.S4> <NAME.Nut> <PNO.P2> <SNO.S4> <QTY.1000>
<COMMON.File> <SNO.S4> <NAME.Dec> <PNO.P2> <SNO.S4> <QTY.2000>
<COMMON.File> <SNO.S4> <NAME.Dec> <PNO.P2> <SNO.S4> <QTY.1000>
```

E. TEST NUMBER 4.

This test demonstrates another retrieve common request.

```
TEST 024[RETRIEVE(TEMP=Part)(PNO,NAME)
        COMMON(PNO,PNO)
        RETRIEVE(TEMP=Ship)(SNO,QTY)]


<COMMON.File> <PNO.P1> <NAME.Idm> <SNO.S3> <QTY.2000>
<COMMON.File> <PNO.P1> <NAME.Idm> <SNO.S3> <QTY.500>
<COMMON.File> <PNO.P2> <NAME.Xyz> <SNO.S1> <QTY.500>
<COMMON.File> <PNO.P2> <NAME.Xyz> <SNO.S2> <QTY.500>
<COMMON.File> <PNO.P2> <NAME.Xyz> <SNO.S4> <QTY.1000>
<COMMON.File> <PNO.P2> <NAME.Xyz> <SNO.S4> <QTY.2000>
<COMMON.File> <PNO.P2> <NAME.Xyz> <SNO.S1> <QTY.2000>
<COMMON.File> <PNO.P2> <NAME.Xyz> <SNO.S1> <QTY.1000>
<COMMON.File> <PNO.P2> <NAME.Xyz> <SNO.S2> <QTY.1000>
```

F. TEST NUMBER 5. This test demonstrates a retrieve-common request using the common value as an integer type.

```
TEST 025[RETRIEVE(TEMP=Ship)(SNO,QTY,PNO)
        COMMON(QTY,QTY)
        RETRIEVE(TEMP=Ship)(SNO,PNO)]
```

<COMMON,File> <SNO,S3> <QTY,500> <PNO,P1> <SNO,S3> <PNO,P1>
<COMMON,File> <SNO,S3> <QTY,500> <PNO,P1> <SNO,S1> <PNO,P2>
<COMMON,File> <SNO,S1> <QTY,500> <PNO,P2> <SNO,S1> <PNO,P2>
<COMMON,File> <SNO,S1> <QTY,500> <PNO,P2> <SNO,S2> <PNO,P2>
<COMMON,File> <SNO,S3> <QTY,500> <PNO,P1> <SNO,S2> <PNO,P2>
<COMMON,File> <SNO,S1> <QTY,1000> <PNO,P2> <SNO,S1> <PNO,P2>
<COMMON,File> <SNO,S1> <QTY,500> <PNO,P2> <SNO,S3> <PNO,P1>
<COMMON,File> <SNO,S2> <QTY,500> <PNO,P2> <SNO,S1> <PNO,P2>
<COMMON,File> <SNO,S1> <QTY,1000> <PNO,P2> <SNO,S2> <PNO,P2>
<COMMON,File> <SNO,S1> <QTY,1000> <PNO,P2> <SNO,S4> <PNO,P2>
<COMMON,File> <SNO,S2> <QTY,1000> <PNO,P2> <SNO,S1> <PNO,P2>
<COMMON,File> <SNO,S2> <QTY,1000> <PNO,P2> <SNO,S2> <PNO,P2>
<COMMON,File> <SNO,S2> <QTY,500> <PNO,P2> <SNO,S2> <PNO,P2>
<COMMON,File> <SNO,S2> <QTY,500> <PNO,P2> <SNO,S3> <PNO,P1>
<COMMON,File> <SNO,S4> <QTY,1000> <PNO,P2> <SNO,S4> <PNO,P2>
<COMMON,File> <SNO,S4> <QTY,1000> <PNO,P2> <SNO,S1> <PNO,P2>
<COMMON,File> <SNO,S2> <QTY,1000> <PNO,P2> <SNO,S4> <PNO,P2>
<COMMON,File> <SNO,S4> <QTY,2000> <PNO,P2> <SNO,S4> <PNO,P2>
<COMMON,File> <SNO,S4> <QTY,1000> <PNO,P2> <SNO,S2> <PNO,P2>
<COMMON,File> <SNO,S1> <QTY,2000> <PNO,P2> <SNO,S3> <PNO,P1>
<COMMON,File> <SNO,S3> <QTY,2000> <PNO,P1> <SNO,S4> <PNO,P2>
<COMMON,File> <SNO,S3> <QTY,2000> <PNO,P1> <SNO,S1> <PNO,P2>
<COMMON,File> <SNO,S3> <QTY,2000> <PNO,P1> <SNO,S3> <PNO,P1>
<COMMON,File> <SNO,S4> <QTY,2000> <PNO,P2> <SNO,S1> <PNO,P2>
<COMMON,File> <SNO,S4> <QTY,2000> <PNO,P2> <SNO,S3> <PNO,P1>
<COMMON,File> <SNO,S1> <QTY,2000> <PNO,P2> <SNO,S4> <PNO,P2>
<COMMON,File> <SNO,S1> <QTY,2000> <PNO,P2> <SNO,S1> <PNO,P2>

## APPENDIX B

## A WALK THROUGH THE USER INTERFACE

In this appendix we present a walk through the user interface. Two aspects of running retrieve-common requests are displayed. The first request is executed from an existing file of traffic untis. The second request has been built on-line. User inputs are in bold letters.

How many backends are there? (1..2....)>**7**

Do you want de-bugging messages printed? (y/n)>**y**
What operation would you like to perform?

(g) - generate database
(l) - load database
(e) - execute test interface
(x) - exit to operating system
(z) - exit and Stop MDBS

**l**
What operation would you like to perform?

(t) - load the template and descriptor files
(r) - mass load a file of records
(x) - exit, return to previous menu

**t**

ENTER NAME OF FILE CONTAINING TEMPLATE INFORMATION:

**tt.f**

ENTER NAME OF FILE CONTAINING THE DESCRIPTORS:

**td.f**

What operation would you like to perform?

(t) - load the template and descriptor files
(r) - mass load a file of records
(x) - exit, return to previous menu

r

NOTE TO THE USER!!!! YOU MUST HAVE LOADED THE TEMPLATES AND DESCRIPTORS FOR A DATABASE, BEFORE ATTEMPTING TO LOAD ANY RECORDS INTO THE DATABASE!!!!

ENTER NAME OF FILE CONTAINING RECORDS TO BE LOADED:

**tr20.f**

TEST 001[INSERT(<TEMP,Part>,<PNO,P1>,<NAME,Idm>,<CITY,Mont>)]
TEST 002[INSERT(<TEMP,Part>,<PNO,P2>,<NAME,Xyz>,<CITY,Sali>)]
TEST 003[INSERT(<TEMP,Part>,<PNO,P3>,<NAME,Nut>,<CITY,Colu>)]
TEST 004[INSERT(<TEMP,Sups>,<SNO,S1>,<NAME,Nut>)]
TEST 005[INSERT(<TEMP,Sups>,<SNO,S2>,<NAME,Nut>)]
TEST 006[INSERT(<TEMP,Sups>,<SNO,S2>,<NAME,Nut>)]
TEST 007[INSERT(<TEMP,Sups>,<SNO,S1>,<NAME,Dec>)]
TEST 008[INSERT(<TEMP,Sups>,<SNO,S3>,<NAME,Nut>)]
TEST 009[INSERT(<TEMP,Sups>,<SNO,S3>,<NAME,Dec>)]
TEST 010[INSERT(<TEMP,Sups>,<SNO,S4>,<NAME,Nut>)]
TEST 011[INSERT(<TEMP,Sups>,<SNO,S4>,<NAME,Dec>)]
TEST 012[INSERT(<TEMP,Ship>,<SNO,S1>,<PNO,P2>,<QTY,500>)]
TEST 013[INSERT(<TEMP,Ship>,<SNO,S2>,<PNO,P2>,<QTY,500>)]
TEST 014[INSERT(<TEMP,Ship>,<SNO,S3>,<PNO,P1>,<QTY,500>)]
TEST 015[INSERT(<TEMP,Ship>,<SNO,S4>,<PNO,P2>,<QTY,1000>)]
TEST 016[INSERT(<TEMP,Ship>,<SNO,S1>,<PNO,P2>,<QTY,1000>)]
TEST 017[INSERT(<TEMP,Ship>,<SNO,S2>,<PNO,P2>,<QTY,1000>)]
TEST 018[INSERT(<TEMP,Ship>,<SNO,S3>,<PNO,P1>,<QTY,2000>)]
TEST 019[INSERT(<TEMP,Ship>,<SNO,S4>,<PNO,P2>,<QTY,2000>)]
TEST 020[INSERT(<TEMP,Ship>,<SNO,S1>,<PNO,P2>,<QTY,2000>)]

What operation would you like to perform?

(t) - load the template and descriptor files
(r) - mass load a file of records
(x) - exit, return to previous menu

**x**

What operation would you like to perform?

(g) - generate database
(l) - load database
(e) - execute test interface
(x) - exit to operating system
(z) - exit and Stop MDBS

**e**

Do you ALWAYS want to wait for responses? (y/n)
> **y**

Enter the type of subsession you want
(r) REDIRECT OUTPUT; select output for answers
(d) NEW DATABASE; choose a new database
(n) NEW LIST; create a new list of traffic units
(m) MODIFY; modify an existing list of traffic units
(s) SELECT; select traffic units from an existing list
(or give new traffic units) for execution
(o) OLD LIST; execute all the traffic units in an existing list
(p) PERFORMANCE TESTING
(x) EXIT; return to generate,load,execute, or exit menu

SELECTION> **s**

Enter the name for the traffic unit file
It may be up to 13 characters long including the .ext.
Filenames may include only one '#' character
as the first character before the version number.

File name> **tRCreq.f**

```
List of executable traffic units
 (0) RETRIEVE(TEMP=Sups)(SNO.NAME)
        COMMON(SNO.SNO)
        RETRIEVE(TEMP=Ship)(PNO.SNO.QTY)

 (1) [RETRIEVE(TEMP=Sups)(SNO,NAME)]

 (2) [RETRIEVE(TEMP=Ship)(PNO,SNO,QTY)]

 (3) [RETRIEVE(TEMP=Part)(PNO.NAME)
        COMMON(PNO.PNO)
        RETRIEVE(TEMP=Ship)(SNO,QTY)]

 (4) [RETRIEVE(TEMP=Part)(PNO,NAME)]

 (5) [RETRIEVE(TEMP=Ship)(SNO,QTY.PNO)
        COMMON(QTY.QTY)
        RETRIEVE(TEMP=Ship)(SNO.PNO)]

 (6) [RETRIEVE(TEMP=Ship)(SNO,QTY.PNO)]
```

/*This section shows how to invoke a predefined retrieve-common request*/

```
    Select Options

    (d)  display the traffic units in the list
    (n)  enter a new traffic unit to be executed
    (num)    execute the traffic unit at [num]
    (x)  exit from this SELECT subsession

    Option> 0
```

```
TEST 023 RETRIEVE(TEMP=Sups)(SNO.NAME)
        COMMON(SNO.SNO)
        RETRIEVE(TEMP=Ship)(PNO.SNO.QTY)


<COMMON.File> <SNO.S1> <NAME.Dec> <PNO.P2> <SNO.S1> <QTY.2000>
<COMMON,File> <SNO.S1> <NAME.Dec> <PNO.P2> <SNO.S1> <QTY.1000>
<COMMON,File> <SNO.S1> <NAME,Nut> <PNO,P2> <SNO.S1> <QTY,500>
<COMMON.File> <SNO.S1> <NAME,Nut> <PNO,P2> <SNO,S1> <QTY,2000>
<COMMON.File> <SNO.S1> <NAME,Dec> <PNO,P2> <SNO,S1> <QTY,500>
<COMMON,File> <SNO.S3> <NAME,Nut> <PNO,P1> <SNO,S3> <QTY,2000>
<COMMON,File> <SNO.S1> <NAME,Nut> <PNO,P2> <SNO.S1> <QTY,1000>
<COMMON,File> <SNO.S2> <NAME,Nut> <PNO,P2> <SNO,S2> <QTY,500>
<COMMON.File> <SNO.S3> <NAME,Nut> <PNO,P1> <SNO,S3> <QTY,500>
<COMMON.File> <SNO.S3> <NAME,Dec> <PNO,P1> <SNO,S3> <QTY,2000>
<COMMON.File> <SNO.S2> <NAME,Nut> <PNO,P2> <SNO,S2> <QTY,1000>
<COMMON.File> <SNO.S2> <NAME,Nut> <PNO,P2> <SNO.S2> <QTY,500>
<COMMON.File> <SNO.S3> <NAME,Dec> <PNO,P1> <SNO,S3> <QTY,500>
<COMMON,File> <SNO.S4> <NAME,Nut> <PNO,P2> <SNO,S4> <QTY,2000>
<COMMON.File> <SNO.S2> <NAME,Nut> <PNO,P2> <SNO.S2> <QTY,1000>
<COMMON,File> <SNO.S4> <NAME,Nut> <PNO,P2> <SNO.S4> <QTY,1000>
<COMMON.File> <SNO.S4> <NAME,Dec> <PNO,P2> <SNO,S4> <QTY,2000>
<COMMON.File> <SNO.S4> <NAME,Dec> <PNO,P2> <SNO,S4> <QTY,1000>
```

/* This section describes the method to build a retrieve-common request*/

Select Options

(d) display the traffic units in the list
(n) enter a new traffic unit to be executed
(num) execute the traffic unit at [num]
(x) exit from this SELECT subsession

Option> **n**

Enter the character for the desired Traffic Unit type.

(r) Request
(t) Transaction (multiple requests)
(f) Finished entering traffic units.

Letter> **r**

Enter the character for the desired next step.

(i) INSERT
(r) RETRIEVE
(u) UPDATE
(d) DELETE
(c) RETRIEVE COMMON

LETTER> **c**

RETRIEVE COMMON Request
First enter the source retrieve request

RETRIEVE Request

Enter responses as you are prompted. You will be prompted first for the predicates of the query, then attributes for the target-list, next for an attribute for the optional BY clause and finally for a pointerfor the optional WITH clause.

When you have finished entering predicates for the query, respond to the ATTRIBUTE> prompt with a <return>.

ATTRIBUTE> **TEMP**

Enter the character for the desired relational operator

(a) = EQUAL
(b) ¬= NOT EQUAL
(c) > GREATER THAN
(d) >= GREATER THAN or EQUAL
(e) < LESS THAN
(f) <= LESS THAN or EQUAL

Letter> **a**

Value> **Sups**

So far your conjunction is
(TEMP=Sups).
Do you wish to 'and' additional predicates to this conjunction? (y/n)
**n**
Do you wish to append more conjunctions to the query? (y/n)
**n**

Begin entering attributes for the Target-List. When you are
through entering attributes respond to the ATTRIBUTE> prompt with <return>.
Do you wish to be prompted for aggregation?
**n**

ATTRIBUTE> **SNO**

ATTRIBUTE> **NAME**

ATTRIBUTE>

COMMON ATTRIBUTE 1> **SNO**

COMMON ATTRIBUTE 2> **SNO**

The request being built is:
RETRIEVE(TEMP=Sups)(SNO.NAME)COMMON(SNO.SNO) 4

Enter the target retrieve

RETRIEVE Request

Enter responses as you are prompted. You will be prompted first for the predicates of the query, then attributes for the target-list, next for an attribute for the optional BY clause and finally for a pointerfor the optional WITH clause.

When you have finished entering predicates for the query, respond to the ATTRIBUTE> prompt with a <return>.

ATTRIBUTE> **TEMP**
Enter the character for the desired relational operator

(a) = EQUAL
(b) /= NOT EQUAL
(c) > GREATER THAN
(d) >= GREATER THAN or EQUAL
(e) < LESS THAN
(f) <= LESS THAN or EQUAL

Letter> **a**

Value> **Ship**

So far your conjunction is
(TEMP=Ship).
Do you wish to 'and' additional predicates to this conjunction? (y/n)
**n**
Do you wish to append more conjunctions to the query? (y/n)
**n**

Begin entering attributes for the Target-List. When you are through entering attributes respond to the ATTRIBUTE> prompt with <return>.
Do you wish to be prompted for aggregation?
**n**

ATTRIBUTE> **SNO**

ATTRIBUTE> **PNO**

ATTRIBUTE> **QTY**

```
The request being processed is:
 RETRIEVE(TEMP=Sups)(SNO.NAME)
   COMMON(SNO.SNO)
   RETRIEVE(TEMP=Ship)(SNO.PNO,QTY)


<COMMON,File> <SNO,S1> <NAME,Dec> <PNO,P2> <SNO,S1> <QTY,2000>
<COMMON,File> <SNO,S1> <NAME,Dec> <PNO,P2> <SNO,S1> <QTY,1000>
<COMMON,File> <SNO,S1> <NAME,Nut> <PNO,P2> <SNO,S1> <QTY,500>
<COMMON,File> <SNO,S1> <NAME,Nut> <PNO,P2> <SNO,S1> <QTY,2000>
<COMMON,File> <SNO,S1> <NAME,Dec> <PNO,P2> <SNO,S1> <QTY,500>
<COMMON,File> <SNO,S3> <NAME,Nut> <PNO,P1> <SNO,S3> <QTY,2000>
<COMMON,File> <SNO,S1> <NAME,Nut> <PNO,P2> <SNO,S1> <QTY,1000>
<COMMON,File> <SNO,S2> <NAME,Nut> <PNO,P2> <SNO,S2> <QTY,500>
<COMMON,File> <SNO,S3> <NAME,Nut> <PNO,P1> <SNO,S3> <QTY,500>
<COMMON,File> <SNO,S3> <NAME,Dec> <PNO,P1> <SNO,S3> <QTY,2000>
<COMMON,File> <SNO,S2> <NAME,Nut> <PNO,P2> <SNO,S2> <QTY,1000>
<COMMON,File> <SNO,S2> <NAME,Nut> <PNO,P2> <SNO,S2> <QTY,500>
<COMMON,File> <SNO,S3> <NAME,Dec> <PNO,P1> <SNO,S3> <QTY,500>
<COMMON,File> <SNO,S4> <NAME,Nut> <PNO,P2> <SNO,S4> <QTY,2000>
<COMMON,File> <SNO,S2> <NAME,Nut> <PNO,P2> <SNO,S2> <QTY,1000>
<COMMON,File> <SNO,S4> <NAME,Nut> <PNO,P2> <SNO,S4> <QTY,1000>
<COMMON,File> <SNO,S4> <NAME,Dec> <PNO,P2> <SNO,S4> <QTY,2000>
<COMMON,File> <SNO,S4> <NAME,Dec> <PNO,P2> <SNO,S4> <QTY,1000>

    Select Options

    (d) display the traffic units in the list
    (n) enter a new traffic unit to be executed
    (num)   execute the traffic unit at [num]
    (x) exit from this SELECT subsession

    Option> x

Enter the type of subsession you want
    (r) REDIRECT OUTPUT; select output for answers
    (d) NEW DATABASE; choose a new database
    (n) NEW LIST; create a new list of traffic units
    (m) MODIFY; modify an existing list of traffic units
    (s) SELECT; select traffic units from an existing list
            (or give new traffic units) for execution
    (o) OLD LIST; execute all the traffic units in an
            existing list
    (p) PERFORMANCE TESTING
    (x) EXIT; return to generate,load,execute, or exit menu

    SELECTION> x
```

What operation would you like to perform?

(g) - generate database
(l) - load database
(e) - execute test interface
(x) - exit to operating system
(z) - exit and Stop MDBS

**z**

# LIST OF REFERENCES


1. Demurjian, S. A., Hsiao, D. K., and Menon, M. J., "A Multi-Backend Database System for Performance Gains, Capacity Growth, and Hardware Upgrade". *Proceedings of the Second International Conference on Data Engineering*, 1986.
2. Tung, H. L., *Design, Analysis, and Implementation of the Primary Operation, Retrieve-Common,of the Multi-Backend Database System (MBDS)*, M.S. Thesis, Naval Postgraduate School, Monterey, California, December 1983.
3. Silberman, B. *Software Portability: A Case Study of the Multi-Backend Database System (MBDS)*, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1986.
4. Wong, A. *Towards Highly Portable Database Systems: Issues and Solutions*, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1986.
5. Bourne, S. R., *The UNIX System*, Addison-Wesley, 1983.
6. Hsiao, D. K. *Advanced Database Machine Architecture*, Prentice-Hall, 1983.
7. Johnson, S. C., *UNIX, Time-Sharing System: UNIX Programmer's Manual*, Bell Telephone Laboratories, Incorporated, 1982.
8. Kelley, A., and Pohl, I., *A Book on C*, The Benjamin/Cummings Publishing Company, 1984.
9. Kernigan, B., and Ritchie, D. M., *The C Programming Language*, Prentice-Hall, 1978.

INITIAL DISTRIBUTION LIST

| | | No. Copies |
|---|---|---|
| 1. | Defense Technical Information Center<br>Cameron Station<br>Alexandria, Virginia 22304-6145 | 2 |
| 2. | Library, Code 0142<br>Naval Postgraduate School<br>Monterey, California 93943-5000 | 2 |
| 3. | Department Chairman, Code 52<br>Department of Computer Science<br>Naval Postgraduate School<br>Monterey, California 93943-5000 | 2 |
| 4. | Curriculum Officer, Code 37<br>Computer Technology Programs<br>Naval Postgraduate School<br>Monterey, California 93943-5000 | 1 |
| 5. | Professor David K. Hsiao, Code 52Hq<br>Computer Science Department<br>Naval Postgraduate School<br>Monterey, California 93943-5000 | 2 |
| 6. | Steven A. Demurjian, Code 52<br>Computer Science Department<br>Naval Postgraduate School<br>Monterey, California 93943-5000 | 2 |
| 7. | CPT Andrew L. Hunt<br>USEUCOM Data Services<br>APO NY 09128 | 3 |