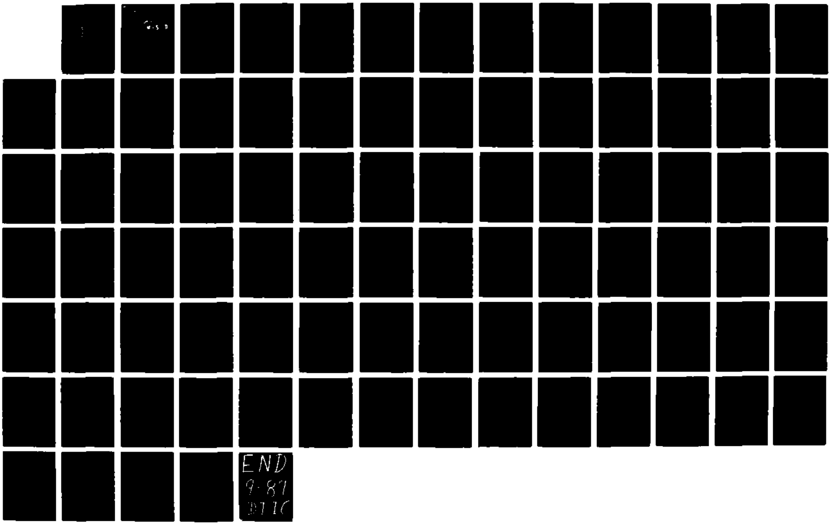
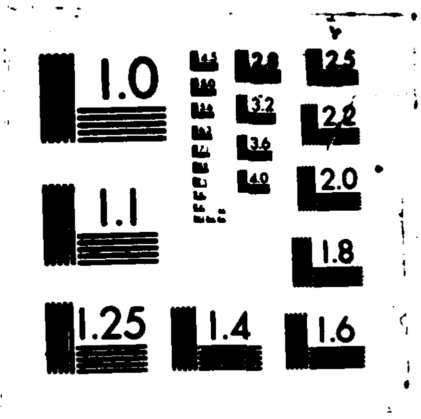
AD-A183 930   BENCHMARKING PREPARATION FOR AND AGGREGATE AND SORTING   1/1
RETRIEVALS IN THE MULTI-BACKEND DATABASE SYSTEM(U)
NAVAL POSTGRADUATE SCHOOL MONTEREY CA   F E KELBE ET AL.
UNCLASSIFIED   JUN 87   F/G 12/7   NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

2

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

BENCHMARKING PREPARATION FOR
AND AGGREGATE AND SORTING RETRIEVALS
IN THE MULTI-BACKEND DATABASE SYSTEM

by

Frank Edward Kelbe

and

Dana Stephen Majors

June 1987

Thesis Advisor: David K. Hsiao

Approved for public release; distribution is unlimited

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

A183930

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION Unclassfied	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY 2b DECLASSIFICATION / DOWNGRADING SCHEDULE	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; Distribution is Unlimited
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b OFFICE SYMBOL (If applicable) Code 52	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	7b ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
8a NAME OF FUNDING / SPONSORING ORGANIZATION 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
BENCHMARKING PREPARATION FOR AND AGGREGATE AND SORTING RETRIEVALS IN THE MULTI-BACKEND DATABASE SYSTEM

12 PERSONAL AUTHOR(S)
Kelbe, Frank Edward and Majors, Dana Stephen

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year Month Day)	15 PAGE COUNT
Master's Thesis	FROM ______ TO ______	1987 June	86

16 SUPPLEMENTARY NOTATION

17 COSATI CODES		
FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Multi-backend Database System; Database; Benchmark, Retrieval, Sorted Retrieval; Aggregate Retrieval, Benchmark Preparation; MBDS; Performance

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

The scope of this thesis is twofold. The first is to provide a methodology for the performance evaluation of the Multi-Backend Database System, MBDS. The second is to describe the implementation and integration for two new database operations, the aggregate retrieval and the sorted retrieval.

The thesis provides the essential tools for the successful evaluation of MBDS. The performance evaluation of MBDS is necessary to validate the performance gains in terms of response-time reduction, and capacity growth in terms of response-time invariance. The implementation and integration of the aggregate retrieval and sorted retrieval provide two advanced data retrieval operations to MBDS. The aggregate retrieval operation allows the user to obtain extremely useful data not inherently available in the

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED □ SAME AS RPT □ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Prof. David K. Hsiao	22b TELEPHONE (Include Area Code) (408) 646-2253 22c OFFICE SYMBOL Code 52Hq

83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

#18 SUBJECT TERMS (continued)

evaluation.

#19 ABSTRACT (continued)

data itself. The sorted retrieval operation allows the user to retrieve data and have it presented in a more meaningful fashion.

Approved for public release; distribution is unlimited

Benchmarking Preparation for and Aggregate and Sorting Retrievals in the Multi-Backend Database System

by

Frank E. Kelbe
Lieutenant, United States Navy
B.S.E.E., University of New Mexico, 1980

and

Dana S. Majors
Lieutenant, United States Navy
B.S., California State University, Sacramento, 1979

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1987

Authors: ______________________
Frank Edward Kelbe

____
Dana Stephen Majors

Approved By: ______________________
David K. Hsiao, Thesis Advisor

____
Steven A. Demurjian, Second Reader

____
Vincent Y. Lum, Chairman,
Department of Computer Science

Kneale T. Marshall
____
Kneale T. Marshall,
Dean of Information and Policy Sciences

ABSTRACT

The scope of this thesis is twofold. The first is to provide a methodology for the performance evaluation of the Multi-Backend Database System, MBDS. The second is to describe the implementation and integration for two new database operations, the aggregate retrieval and the sorted retrieval.

The thesis provides the essential tools for the successful evaluation of MBDS. The performance evaluation of MBDS is necessary to validate the performance gains in terms of response-time reduction, and capacity growth in terms of response-time invariance. The implementation and integration of the aggregate retrieval and sorted retrieval provide two advanced data retrieval operations to MBDS. The aggregate retrieval operation allows the user to obtain extremely useful data not inherently available in the data itself. The sorted retrieval operation allows the user to retrieve data and have it presented in a more meaningful fashion. (Theses)

TABLE OF CONTENTS

ACKNOWLEDGEMENT

This thesis is part of ongoing database systems research being conducted at the Laboratory for Database Research at the Naval Postgraduate School, Monterey, California, 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.

We would like to thank the following people who have helped us complete this thesis:

Dr. David K. Hsiao, for the guidance, wisdom and motivation he provided.

Dr. Steve Demurjian, who's technical expertise was continually called upon in order to complete this thesis.

I. INTRODUCTION

Organizations have the need to perform fast, accurate, efficient, and economical information processing. Database systems consisting of both hardware and software, better known as database management systems (DBMS), have been designed to meet these needs. Many variations of database systems have recently entered the marketplace, each tailored to meet specific processing requirements. Research in the computer science community has been pursuing several new approaches to satisfy today's growing information needs. In this chapter we begin by reviewing the research efforts in DBMS, to provide a background for the thesis. Next we present a brief discussion on the Multi-Backend Database System. Third, we outline the motivation for the work presented in this thesis. Finally, we review the organization of the thesis.

A. THE BACKGROUND

In this section, we focus on the impact of various database system architectures on their hardware upgrade. Database system hardware must be upgraded due to either the performance degradation of the system software or the advances in technology. We first review the three approaches to database systems. Next, we discuss in more detail the software multiple-backend database approach. Finally, we discuss a specific multiple-backend system, the Multi-Backend Database System, or MBDS. The discussion focuses on three aspects of MBDS; the design considerations, the performance and capacity growth claims, and the system configuration.

1. Three Database-System Approaches

As identified by Hsiao [Ref. 1], research has produced three database-system approaches: 1) the traditional mainframe-based system, 2) the software single-backend system, and 3) the software multiple-backend system. The mainframe-based approach is characterized by the database system residing on the mainframe computer as an applications program. The database system must share the computer's resources with other application programs residing on the computer.

In the single-backend approach, the database system resides on a dedicated backend computer. The general-purpose computer (termed the host) provides the interface between the user and the database system. All database management services are provided by the backend computer via the host. The database system, residing on the backend, has exclusive access to all of the resources on the backend computer.

The software multiple-backend system is an extension of the single-backend concept. There is one or more controller computers connected to multiple, backend computers. The interface to the database between the user and the host is through the controller computer(s). Each backend contains a portion of the database, and maintains exclusive access to the data. The software multiple-backend system is designed to overcome both performance and upgrade problems normally experienced with the traditional and the single-backend systems. This system is more unconventional than the first two, and is a new kind of database system

2. Software Multiple-Backend Database Computers

In the software multiple-backend approach, the database system is not mainframe-based, and each database system consists of at least one controller and two or more backends. A communications network is used to interconnect the backend and controller systems. This approach differs from the single-backend approach in that the database is not physically located on a single backend. Instead, the database is distributed across all of the multiple backend computers. As to functionality, the controller is responsible for 1) the communication with the hosts (or terminals), 2) the scheduling and control of transactions being executed by the backends, 3) the correlation of the data for each transaction from all of the backends, and 4) the routing of the responses back to the user. The backend software is replicated across all of the backend computers. The functionality of the system requires each backend to be responsible for 1) the management and execution of database transactions, 2) permitting concurrent access to data via parallel processing of transactions, 3) processing the database information stored on the disk, and 4) communicating with other backends and the controller to pass information and results. Each backend is also responsible for executing the required primary database operations such as retrieve, insert, delete, and update.

Unlike the other two approaches, the software multiple-backend approach stresses large-capacity and high-performance database management. The capacity growth and performance gains are now directly related to the number of backends in the system. An increase in the number of backends for a given system can result in both increased capacity and performance.

3. The Multi-Backend Database System (MBDS)

The Laboratory for Database Research at the Naval Postgraduate School has developed a prototype software multiple-backend system, known as MBDS [Refs. 2,3,4]. One minicomputer or microcomputer serves as the controller, while multiple microcomputers and their associated disk systems serve as the backends. The controller and all backends are interconnected by a high-speed broadcast bus. Together, the three subsystems, controller, backends, and broadcast bus, constitute a system specifically designed to overcome the performance and capacity growth problems normally experienced by traditional database systems. The data in the MBDS system is evenly distributed across all of the backend disk systems. A user transaction may therefore be executed simultaneously by all backends.

a. Design Considerations

The design of MBDS, proposed by Menon [Ref. 5], has been influenced by three primary objectives; 1) performance gains in terms of response-time reductions, 2) capacity growth in terms of the response-time invariance, and 3) system expandability. The first goal enables the multiplicity of the backends to be directly related to the capacity growth of the system in terms of the response-time invariance. By increasing the number of backends proportionally to the increase of transaction responses, MBDS produces invariant response times for user transactions. The second goal permits the multiplicity of the backends of MBDS to be directly related to the performance gains of the system in terms of response-time reduction. By increasing the number of backends while the size of the database and the size of the transaction responses remain constant, the MBDS system produces a reciprocal reduction in the response times for user transactions. The third goal has been met, in terms of the ease in adding new hardware (i.e., backends) to the system, and in configuring the existing software. When a new backend is added to the system, the software is replicated to the new backend, and the database is redistributed.

b. Performance and Capacity Growth Claims

Performance problems and upgrade issues have always been an obstacle in traditional mainframe-based systems and software single-backend systems. The software multiple-backend approach attempts to overcome these problems through specialization of the database operations on multiple, dedicated backends.

The two goals of the Multi-Backend Database System are to overcome upgrade and performance problems normally associated with traditional systems. Expansion is made easy by replicating the software on all backends in a system. Expansion to a system simply requires the necessary hardware for a new backend. The software on the new backend is identical to the existing backends. If a database system is extended by adding new backends while keeping the size of the database constant, a reciprocal decrease in the response times for given transactions occurs. Second, if the system is extended by adding a number of backends proportional to the increase in transaction responses, the system produces invariant response times for given transactions. This allows a database to grow with no sacrifice in performance.

The first goal directly relates the multiplicity of the number of backends in the system to the performance gains of the system in terms of the response-time reduction. Response-time reduction is a measure of the time reduction associated with processing a given set of requests on a system with multiple backends, as compared to a single-backend system. The second goal relates the multiplicity of the backends to the capacity growth of the system in terms of response-time invariance. Response-time invariance is the change in the response time of a request, when the request is processed in a single-backend system with a response set of x records, as compared to processing the same transaction in a system with m backends and a response set of mx records [Ref. 6]. A response set is the set of responses returned by the backend(s) to the user for a given transaction. The size of the response set is determined by the size of the database (i.e., a given request produces more responses in a large database.) The definition of response-time invariance must therefore be restated as the amount of change in the response time of a given request, when the request is processed in a single-backend system with a database size of x records, as compared to processing the same request in a system with m backends and a database size of mx records.

c. The MBDS System Configuration

The MBDS hardware configuration at the Laboratory for Database Research consists of eight ISI (Integrated Solutions Incorporated workstations) microcomputers. All systems utilize the 4.2 BSD Unix operating system. One workstation functions as the controller, leaving seven workstations to act as backends. Each workstation is based on the Motorola 68020 CPU, featuring 16 megabytes of virtual memory space per process. The controller system has four Mbytes of main memory, while each of the backends has two Mbytes of main memory. Each backend has a pair of dedicated Control Data Corporation Winchester-type drives: one drive with a capacity of 100 Mbytes dedicated to the operating system, and a second drive dedicated to the database system, having a capacity of 500 Mbytes. The system is connected by way of an Ethernet broadcast bus, capable of transferring data at a rate of 10 megabits per second.

B. SCOPE OF THE THESIS

The scope of this thesis is twofold. The first scope is to provide a general methodology that may be used in the performance evaluation of a database computer, namely the Multi-Backend Database System, MBDS. The scope is also to provide the necessary tools for evaluating the system. As previously discussed in Section A, the two claims of the multi-backend design are performance gains in terms of response-time reduction, and capacity growth in terms of response-time invariance. The validation of these claims is of our primary concern. We present a detailed discussion concerning database configuration considerations. Included in the discussion are references to preferred test database sizes and record sizes. We also discuss in detail the relationships between several internal parameters within the MBDS software. The determination of these parameters is critical to the successful performance evaluation of MBDS. The discussions also contain recommendations concerning test set generation and system configurations.

The second scope of the thesis is to describe the implementation and integration considerations for two new database operations. The current data language implemented on MBDS provides for five primary database operations. These five operations are Insert, Delete, Update, Retrieve, and Retrieve-Common. In addition, two types of retrieval options, while initially designed, have never been implemented. The first operation is the aggregate retrieval. The second operation is the sorted retrieval, or by-clause. The combination of retrieval with aggregation and sorting, is also considered. These new options allow the user to process the retrieved data into a more useful form. The operations have been implemented and integrated into the existing MBDS software to provide more powerful database access operations.

C. ORGANIZATION OF THE THESIS

In addition to this introduction, the thesis is divided into five chapters. In Chapter II we provide an overview of the Multi-Backend Database System, and give the necessary background material on MBDS used in the context of the thesis. In Chapter III we describe a general methodology for the performance evaluation of MBDS. A discussion of relevant conditions and system configurations is given. Since

Chapters IV and V are similar in scope, for both we provide an in-depth discussion of the implementation and integration of two new functions into MBDS. In Chapter IV we address the addition of the aggregate retrieval operation, while in Chapter V we address the sorted retrieval, or by-clause, operation. Finally, in Chapter VI we present a summary and the conclusions of the thesis, as well as some insight into the problems of integration that were encountered. Also included is a brief discussion on possible future work to be conducted.

II. THE MULTI-BACKEND DATABASE SYSTEM (MBDS)

In this chapter, we discuss the required background material on MBDS that is essential for reading this thesis. First, we present an overview of the data model of MBDS, the attribute-based data model, which allows users to specify the structure of the database. Next, we present a description of the internal directory structure of MBDS. Third, we review the attribute-based data language (ABDL), with a description of the different types of database operations. Fourth, we overview the system structure of MBDS, focusing on how the software is partitioned by functionality in the controller and backends. Finally, we provide a description of the MBDS message format, and a complete listing of the message types.

A. THE ATTRIBUTE-BASED DATA MODEL

MBDS is based on the attribute-based data model proposed in [Ref. 7], extended in [Ref. 8], and studied in [Ref. 9]. In the attribute-based data model, data is considered in the following constructs: database, file, record, attribute-value pair, keyword, attribute-value range, directory keyword, non-directory keyword, directory, record body, keyword predicate, and query. Informally, a database is a collection of files. Each file contains a group of records characterized by a unique set of keywords. Each record is composed of two parts; attribute-value pairs, or keywords, and the record body. An attribute-value pair is a member of the Cartesian product of the attribute name and the value domain of the attribute. As an example, the attribute-value pair <POPULATION, 30000> has a value of 30000 for the population attribute. A record contains at most one attribute-value pair for each attribute defined in the database. Directory keywords for a record (or a file) are either the attribute-value pairs or their attribute-value ranges that are kept in a directory for identifying the records (files). Those attribute-value pairs which are not kept in a directory are appropriately termed non-directory keywords. The record body is the rest of the record, and is normally textual information. An example record is shown below.

```
    (<FILE,USCensus>,<CITY,Carmel>,<POPULATION,15000>,
                   {Temperate climate})
```

The angle brackets, <,> enclose an attribute-value pair, i.e., keyword. The curly brackets, {,} include the record body. The record is enclosed in the parenthesis. By convention, the first attribute-value pair of all records of a file is the same. The attribute is normally FILE, and the value is the appropriate file name.

The records of the database may be identified by keyword predicates. A keyword predicate is a tuple consisting of a directory attribute, a relational operator (=, !=, <, <=, >, >=), and an attribute value. An example of a keyword predicate, or more specifically a less-than predicate, would be POPULATION < 25000. These keyword predicates, combined in disjunctive normal form, comprise a query of the database. The query

```
(FILE = USCensus and CITY = Monterey) or
(FILE = USCensus and CITY = Carmel)
```

will be satisfied by all records of the USCensus file with the CITY of Monterey or Carmel. The parenthesis bracketing the conjunction are simply for clarity.

B. THE DIRECTORY STRUCTURE

To manage the database (often referred to as user data), MBDS uses directory data. The directory has the following constructs: attributes, descriptors, and clusters. An attribute is used to represent a category of the user data; e.g., POPULATION is an attribute that corresponds to actual populations stored in the database. A descriptor is used to describe a range of values that an attribute can have; e.g., (10001 < POPULATION < 15000) is a possible descriptor for the attribute POPULATION. The descriptors that are defined for an attribute, e.g., population ranges, are mutually exclusive. The notion of a cluster may now be defined. A cluster is a group of records such that every record in the cluster satisfies the same set of descriptors. For example, all records with POPULATION between 10001 and 15000 may form one cluster whose descriptor is the one given above. In this case, the cluster satisfies the set of a single descriptor.

In reality, a cluster tends to satisfy a set of multiple descriptors. The directory is organized in three tables: the attribute table (AT), the descriptor-to-descriptor-id table (DDIT), and the cluster-definition table (CDT). The attribute table maps directory attributes to the descriptors defined on them. A sample AT is depicted in Figure 2.a. The descriptor-to-descriptor-id table maps each descriptor to a unique descriptor id. A sample DDIT is given in Figure 2.b. The cluster-definition table maps descriptor-id sets to cluster-ids. Each entry consists of the unique cluster id, the set of descriptor ids whose descriptors define the cluster, and the ids of the records in the clusters. A sample CDT is shown in Figure 2.c. Thus, to access the user data, MBDS must first access directory data via the AT, DDIT, and CDT.

C. THE DATA MANIPULATION OPERATIONS

The attribute-based data language (ABDL), as defined by Banerjee [Ref. 10] and extended by Tung [Ref. 11], is the data language of MBDS. ABDL supports five primary database operations, INSERT, DELETE, UPDATE, RETRIEVE, and RETRIEVE-COMMON. This section provides a discussion of the two classes of operations, the database modification operations, and the database access operations.

A request in ABDL is defined as a primary operation with a qualification. A qualification specifies the part of the database that is to be operated on. A transaction is a group

Attribute	DDIT Entry
POPULATION	D11
CITY	D21
FILE	D31

Figure 2a. An Attribute Table (AT)

0 ≤ POPULATION ≤ 25000	D11
25001 ≤ POPULATION ≤ 100000	D22
100001 ≤ POPULATION ≤ 250000	D13
250001 ≤ POPULATION ≤ 1000000	D14
CITY = Monterey	D21
CITY = Carmel	D22
CITY = London	D23
CITY = Toronto	D24
FILE = CanadaCensus	D31
FILE = USCensus	D32

(Dij: Descriptor j for attribute i)

Figure 2b. A Descriptor-to-Descriptor-Id Table (DDIT)

Id	Desc-Id Set	Rec-Id
C1	{D11,D21,D32}	A1,A2
C2	{D14,D22,D32}	A3
C3	{D12,D23,D32}	A4
C4	{D14,D24,D31}	A5

Figure 2c. A Cluster-Definition Table (CDT)

of two or more requests. The five primary database operations may be categorized into two types of requests; operations that modify the database, and operations that do not modify the database, or retrieval operations. ABDL provides five seemingly simple database operations, which are nevertheless capable of supporting complex and comprehensive transactions. Following is an informal presentation of the two classes of operations and the five request types.

1. Database Modification Operations

Database modification operations are those operations which modify the database in some way when performed. In ABDL, the modification operations are INSERT, UPDATE, and DELETE. The INSERT request is used to insert a new record into the database. The qualification of an INSERT request is a list of keywords and a record body. The following example,

```
INSERT (<FILE, USCensus>,<CITY, Carmel>,<POPULATION,1500>)
```

is a request that inserts a new record with no record body into the USCensus file for the city of Carmel with a population of 15000.

An UPDATE request is used to modify existing records in the database. The qualification of an UPDATE request consists of two parts. The first part is the query, which specifies which records in the database are to be modified. The second part is the modifier, which specifies how the records being modified are to be changed. In the following example,

```
UPDATE ((FILE = USCensus) and (POPULATION > 50000))
       <POPULATION = POPULATION + 7500>
```

the request modifies all records of the USCensus file where the population is greater than 50000. Those records which match the query will have the POPULATION value increased by 7500. In this example, ((FILE = USCensus) and (POPULATION > 50000)) is the query, and (POPULATION = POPULATION + 7500) is the modifier.

A DELETE request is used to permanently remove one or more records from the database. The qualification of a DELETE request is a query. All records which match the query in the database are deleted. The following example,

```
DELETE((FILE = USCensus) and (POPULATION > 45000))
```

is a request that removes all records in the USCensus file whose population value is greater than 45000.

2. Database Access Operations

Database access operations do not modify the database in any way when performed. Data is only retrieved for examination from the database. The database may be accessed in several ways. This section provides brief descriptions and examples of each of the access operations.

The RETRIEVE request is used to retrieve records from the database. The qualification of a retrieve request consists of a query, a target-list, and an optional by-clause. The query specifies which records are to be retrieved. The target-list consists of a list of output attributes. The target-list may also contain aggregate operations, i.e., AVG, COUNT, SUM, MIN, MAX, on one or more of the output attributes. The optional by-clause may be used to sort the output data in relation to one of the specified output attributes. The following RETRIEVE example,

```
RETRIEVE ((FILE = USCensus) and (POPULATION > 30000))
         (CITY)
```

retrieves the city names of all records in the USCensus file with populations greater than 30000. The query is (FILE = USCensus) and (POPULATION > 30000) and CITY is the target-list. An example of a retrieve using an aggregate operator would be,

```
RETRIEVE (FILE = USCensus)(COUNT(CITY))
```

which would return a count of all the cities within the USCensus file. In this example, the query is simply (FILE = USCensus), and the target-list (COUNT(CITY)) is composed of only the aggregate operator COUNT. The following example utilizing a by-clause,

```
RETRIEVE ((FILE = USCensus) and (POPULATION < 100000))
         ((CITY, POPULATION) BY CITY)
```

would retrieve all of the cities with a population less than 100000, and then present the cities and the corresponding populations. The data is shown sorted in ascending order by city name. The query in this example is ((FILE = USCensus) and (POPULATION < 100000)) and the target-list is (CITY, POPULATION) BY CITY. The BY CITY in the target list specifies that the data is to be sorted by city name.

Finally, the RETRIEVE-COMMON request is used to merge two files by common attribute-values. Logically, the RETRIEVE-COMMON request can be considered as two requests that are processed serially in the following form.

```
RETRIEVE (query-1)(target-list-1)
COMMON (attribute-1, attribute-2)
RETRIEVE (query-2)(target-list-2)
```

The common attributes are attribute-1 (associated with the first retrieve request) and attribute-2 (associated with the second request). The next example is a RETRIEVE-COMMON request,

```
RETRIEVE ((FILE = USCensus) and (POPULATION > 200000))(CITY)
COMMON (POPULATION, POPULATION)
RETRIEVE ((FILE = CanadaCensus) and (POPULATION > 200000))(CITY)
```

which finds all records in the CanadaCensus file with population greater than 200000, finds all records in the USCensus file with population greater than 200000, identifies records of respective files whose population figures are common, and returns the two city names whose cities have the same population figures.

D. THE PROCESS STRUCTURE

In addition to the two communication processes, get-net, which sends a message over the network, and put-net, which receives a message from the network, there are other processes in MBDS. Currently, MBDS does not communicate with a host computer. This absence requires that the test-interface, the process used to interact with MBDS, be placed within the MBDS controller. The software of a backend is complete, and is capable of performing all of the primary database operations. An overview of the MBDS process structure, both controller and backend, may be seen in Figure 2.d.

1. The Processes of the Controller

In addition to the communications and test-interface processes, the controller consists of three additional processes: request preparation, insert information generation, and post processing. Request preparation receives, parses, and formats a request (transaction) before sending the formatted request (transaction) to the directory-management process in each backend. Insert information generation is used to provide additional information to the backends when an insert request is received. Since the user data is distributed, the insert occurs only at one of the backends. Thus the controller must determine the backend at which the insert will occur, along with certain directory information. Post processing is used to collect all the results of a request (transaction) and forward the results to the user.

2. The Processes of Each Backend

In addition to the communication processes, each backend is composed of four other processes. They are, of course, different from the controller processes. They are

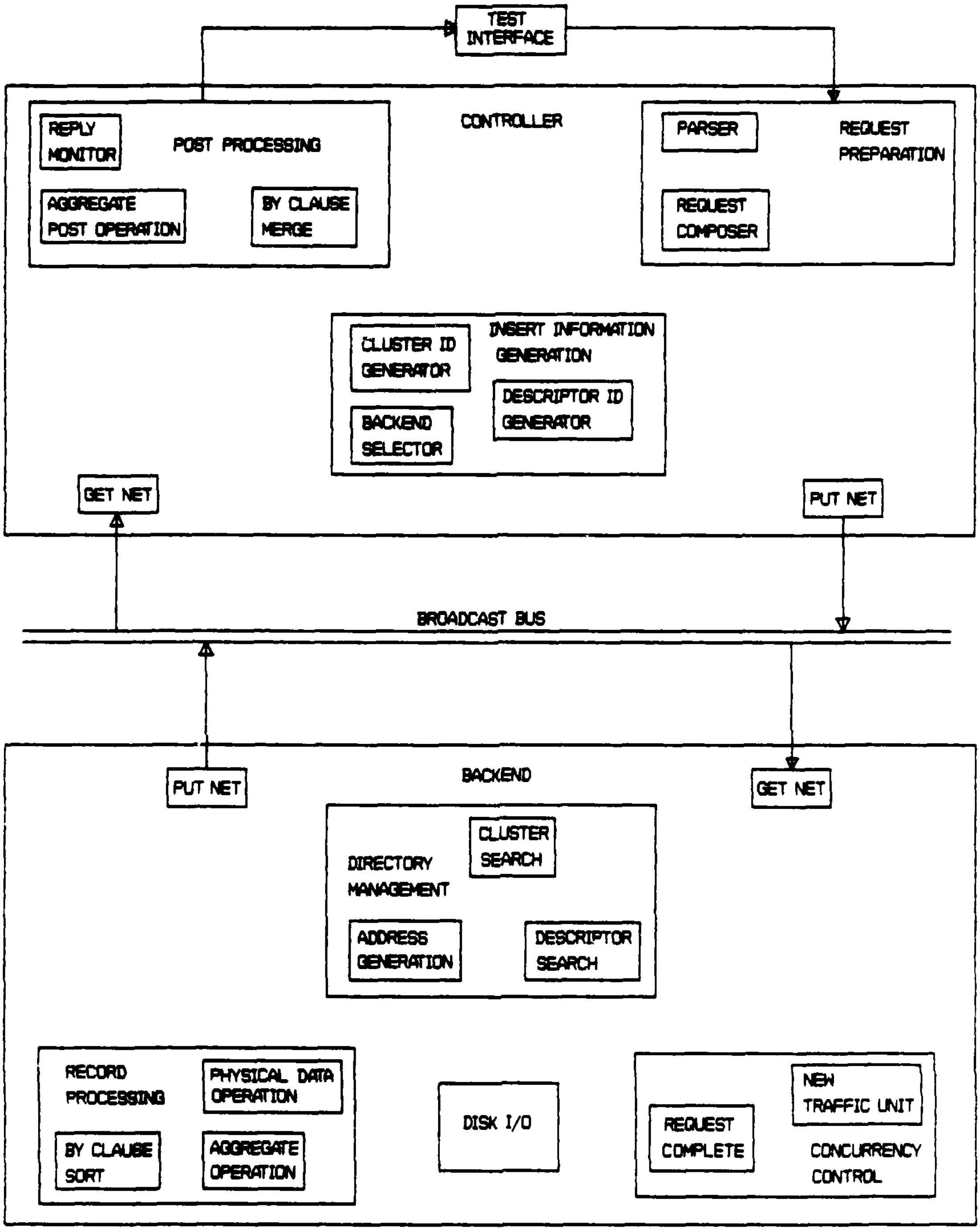


Figure 2.d The MBDS Process Structure

directory management, concurrency control, disk I/O, and record processing. Directory management performs the search of the directory tables to determine the secondary storage addresses necessary to access the clustered records. More specifically, directory management controls the execution of a request at a backend, and accesses the secondary-storage-based directory tables, i.e., AT, DDIT, and CDT. By traversing the directory tables for a request, directory management is able to determine the disk address where the relevant data is stored. (We recall that the disk addresses are in the CDT.) These disk addresses are then sent to record processing which accesses the clustered records. Concurrency control is used to arbitrate the access of the directory data and user data. Since new descriptors, new clusters, and new secondary storage addresses may be defined dynamically, concurrency control is used to ensure the consistency of both the directory data and the user data. Record processing performs the access and modification of the database by issuing I/O requests to the disk I/O process, operating on the retrieved data for retrieval, aggregation, and sorting, and altering the database for modification operations. Finally, the disk I/O process is used to issue read and write requests in a synchronous fashion, and employs a scheduling algorithm to optimize disk access.

E. THE MBDS MESSAGE TYPES AND FORMAT

In this section, we briefly describe the MBDS message-passing facilities first described in [Ref. 7]. MBDS utilizes one general message format, as shown in Figure 2.e.

Message Type	(a numeric code).
Message Sender	(a numeric code).
Message Receiver	(a numeric code).
Message Text	(an alphanumeric field terminated by an end-of-message marker).

Figure 2.e The General Message Format

This same message format is used for each of the three message passing facilities, namely, messages within the controller, messages within each backend, and messages between computers. The message type is one of the 36 message types contained within the MBDS message-passing facilities. These messages are shown in Figure 2.g. The message sender and receiver in Figure 2.e can be any one of the 12 processes in either the controller or the backend.
The message text is the actual data being sent in the message. Figure 2.g provides a complete list of the message types. The figure includes a column for the source, destination, and path for each message type. The key to the codes used in each of the columns is given in Figure 2.f.

Source or Destination Designation	Path Designation
HOST : Host Machine (Test Interface)	H : Host
REQP : Request Preparation	C : Controller
IIG : Insert Information Generation	C : Controller
PP : Post Processing	C : Controller
DM : Directory Management	B : A Backend
RECP : Record Processing	B : A Backend
CC : Concurrency Control	B : A Backend

Figure 2.f The MBDS Message Types

As an example, consider message 12, Backend Aggregate Operator results, from Figure 2.g. The entry in the source column is RECP and can be found in Figure 2.f corresponding to the source, record processing. Similarly, the entry for DEST in Figure 2.g is PP, corresponding to the destination of post processing from Figure 2.f. The key to the two letters in the PATH column may also be found in Figure 2.f. A path of BC is found to be an inter-computer path from a backend to the controller. Each message type has one distinct source, destination, and path to follow.

MESSAGE-TYPE NUMBER AND NAME	SOURCE	DEST	PATH
1 Traffic Unit	HOST	REQP	HC
2 Request Results	PP	HOST	CH
3 Number of Requests in a Transaction	REQP	PP	C
4 Aggregate Operators	REQP	PP	C
5 Requests With Errors	REQP	PP	C
6 Parsed Traffic Unit	REQP	DM	CB
7 New Descriptor Id	IIG	DM	CB
8 Backend Number	IIG	DM	CB
9 Cluster Id	DM	IIG	BC
10 Request for a New Descriptor Id	DM	IIG	BC
11 Backend Results for a Request	RECP	PP	BC
12 Backend Aggregate Operator results	RECP	PP	BC
13 Backend By-Clause Results	RECP	PP	BC
14 Sorted Results to Post Processing	RECP	PP	BC
15 Record That has Changed cluster	RECP	REQP	BC
16 Results of a Retrieve Caused by an Update	RECP	REQP	BC
17 Descriptor Ids	DM	DMs	BB
18 Request and Disk Addresses	DM	RECP	B
19 Changed Cluster Response	DM	RECP	B
20 Fetch	DM	RECP	B
21 Old and New Values of Attribute Being Modified	RECP	DM	B
22 Type-C Attributes for a Traffic Unit	DM	CC	B
23 Desc-Id Groups for a Traffic Unit	DM	CC	B
24 Cluster Ids for a Traffic Unit	DM	CC	B
25 Release Attribute	DM	CC	B
26 Release all Attributes for an Insert	DM	CC	B
27 Release Descriptor Id Groups	DM	CC	B
28 Attribute Locked			
29 Descriptor-Id Groups Locked	CC	DM	B
30 Cluster Ids Locked	CC	DM	B
31 Generated Inserts Completed	RECP	REQP	BC
31 Generated Inserts Completed	REQP	DM	CB
31 Generated Inserts Completed	DM	RECP	BC
32 Request Id of a Completed Request	RECP	CC	B
33 Update Request has Completed	RECP	DM	B
33 Update Request has Completed	RECP	DM	B
34 Source Retrieve-Common has Completed	DM	CC	B
35 Notification of a Retrieve-Common Request	REQP	RECP	CB
36 Target Retrieve-Common Records	RECP	RECPS	B

Figure 2.g The MBDS Message Types

III. PERFORMANCE EVALUATION

In this chapter, we detail a general methodology that may be used in the performance evaluation of a database system. In the first part of the chapter, we present a performance evaluation methodology for MBDS. The benchmark strategy focuses on collecting the response time of requests (transactions) that are processed by the system. In the same section, we describe system dependent considerations along with some remarks on the utilization of a computer-aided benchmarking system (CABS), detailed in [Refs. 6,12]. In the second part of this chapter, we discuss database system configuration considerations for benchmarking. The MBDS system's internal constants and parameters are discussed in detail, and some example configurations are provided.

A. A PERFORMANCE MEASUREMENT METHODOLOGY

Performance evaluation, or benchmarking, involves the design and generation of test transactions and test databases for prototype database systems. The tests must be thorough and be able to establish standards (i.e., benchmarks) for the performance or throughput of the database system. The methodology presented in this section is developed to evaluate a ɩpecific class of database systems, namely, multiple-backend database systems. The key concern in the benchmarking of a database system is the specification of the workload. The workload of a database system is characterized by three models that are hierarchically dependent: a model of the machine, a model of the database, and a model of the application. Therefore, to adequately develop a fair and unbiased benchmark set, the workload must be machine-independent, database-independent, and application-independent. To achieve machine-independence, the benchmark is constructed without bias toward any particular hardware organization or software architecture. To achieve database-independence, the benchmark database is independent of the database model of the real-world database. And, to achieve application-independence, the benchmark applications are generic.

In the remainder of this section, we first provide a brief description of the two types of performance measurements, and why each type is necessary. Next, we describe some of the system configuration modifications that were required to facilitate the performance evaluation of MBDS. Finally, a discussion on the utilization of CABS in the generation of the test benchmark set is provided.

1. Two Types of Performance Measurement

There are two types of methodologies applicable to the benchmarking of a database system. The first is the internal performance measurement methodology, characterized by the fine granularity of the measurements produced. The second methodology is the external performance measurement methodology, providing rough (relatively) measurements of overall system performance.

The goal of the internal performance measurement methodology is to provide methods and tools to enable us to better understand the target system by measuring specific aspects of that system. A complete understanding of how the system performs internally may lead to design modifications or to fine-tuning of the system for better performance. The tools should be fully integrated into the system, leaving them transparent to the user. The methodology relies on checkpoints internal to the database system software. A *checkpoint* is defined as a procedural invocation inserted into the system's flow of control to call the performance measurement routines used for the data collection. The adding of checkpoints into the system introduces additional system overhead. The checkpoint software places additional demands on the system by requiring the resources for data storage, message passing, and information processing relating to the checkpointing data.

The goal of external performance measurement is to provide a collection of methods and tools which enable us to measure the system as a whole. Using this methodology, we can measure the total work being done by the database system. The focus of external measurements is on the response time of the system, i.e., the elapsed time between the issuance of a request and the receipt of the response to the request. Whereas internal performance measurement has been shown to be useful in the microscopic examination of the work being performed by a system, external performance measurement provides a macroscopic view of the work being performed by a system. The external performance measurement software should have negligible overhead, i.e., the response time with external performance measurements being performed should be the same as the response time without the measurements being performed. The reason the overhead is negligible is that only two timing checkpoints need to be made. These checkpoints are placed at the beginning of a request and the end of the response to the request, thus providing the elapsed time of the response for a request. The checkpoints are placed at a very high level to ensure a complete measurement of the total elapsed time.

2. Modifications to MBDS software

Several minor modifications to the existing MBDS software are required to allow benchmarking of the system. This section first describes those modifications, and then the limitations imposed.

a. Specific Modifications

The Multi-backend Database System originally utilized a VAX-11/780 (VMS OS) as the controller, and two PDP-11/44s (RSX-11M OS) as backends. As described in Chapter I., the current configuration utilizes ISI workstations for both the controller and the backends. The change from the

VMS operating system to the use of ISI workstations and the 4.2 BSD Unix operating system required a change in the timing software of MBDS. The basic operation of the timers, as previously discussed, remained unchanged because the changes were operating system dependent. The scope of the changes was limited to changing the calls to system supplied functions. These system function calls were those concerned with retrieving times from the hardware's internal system clock.

There was a change in the granularity of the times available when utilizing the system clock. The previously available granularity was limited to time units in terms of hundredths of a second. This was acceptable for external timing measurements, but unacceptable when performing internal timing measurements. The modifications provided the software with a microsecond time units. Although this is a very fine granularity, and at first appears to be very useful for obtaining precise internal performance measurements, we found that this was not the case. When using times with rough granularity such as hundredths of a second, the execution of the software in terms of function calls, system calls, etc., is negligible. When the granularity becomes as fine as microseconds, those previously negligible times now become a factor. The time required for calls now impacts the results of the internal measurements, thereby providing incorrect results. For this reason, although microsecond time units were available from the system, only three decimal digits, (thousandths of a second) were actually utilized. The last two digits of precision were discarded.

The user interface required modification to allow the presentation of the external timing results. The previous configuration presented the user with only the elapsed time. The modification added the additional presentation of start time, stop time, and the number of buffers of data that were received in the course of the response to the transaction. The number of buffers was shown to allow the evaluator to determine how much data was actually received from the backends in comparison to the expected amount of data. As we discuss in Section C, this allows the evaluator to compare the results from two requests, each returning the same number of buffers, but accessing different amounts of data. This comparison can provide a rough estimate of the system overhead in terms of message passing.

b. Limitations

There are several specific limitations on the existing system that impact on the benchmarking of the system. The first of these limitations is the internal performance measurement software. The performance measurement system places additional demands on the MBDS system message-passing software. The message-passing routines of the MBDS backends are not designed to handle the transfer of, normally, 200 internal performance-measurement messages from a backend to the controller. There is not sufficient space available to store the information required to access this many messages. MBDS contains all the necessary software for internal performance measurement, but, for the reasons just stated, does not utilize the functions. When the internal performance measurement functions are required, the MBDS system is easily extended to meet the demands.

The second limitation on the system is due to the message-passing protocols utilized in MBDS. Messages sent to the backends from the controller, and messages between backends are broadcast over the Ethernet. The problem with this protocol is at the receiving end. Because each backend is waiting for broadcast messages, it accepts all incoming messages, assumes the message is destined for that backend, and processes it. This presents a problem if the message is from another system, process, etc. The software is designed around the premise that MBDS is the only user of the network. This assumption is very often invalid. For this reason, attempts must be made to isolate MBDS and the network from the rest of the 'world' when benchmarking is being performed. This prevents the unnecessary overhead of having to process, and ignore unwanted messages found on the network.

The final system limitation is specific to the hardware and associated operating system being utilized. Since MBDS has multiple processes executing on the controller and backends, the operating system must obviously support multi-tasking. A multi-tasking capability normally has associated with it the virtual memory concept, the ability to page data from primary to secondary memory, as well as the ability to swap processes to and from secondary memory. This can present inconsistencies if the operating system is swapping MBDS processes to secondary memory during the benchmarking process. To preclude this problem, it becomes necessary to prevent other users from utilizing any of the systems being used by MBDS and to also be able to force the MBDS processes of both the controller and backends to be memory resident.

3. A Computer-Aided Benchmarking System

A Computer-Aided Benchmarking System (CABS) as described in [Refs. 6,12] is available for use in the performance evaluation of MBDS. The original design factors utilized in the design and implementation of CABS were presented in [Ref. 13]. This section presents some remarks on its effectiveness in relation to the benchmarking of MBDS, an overview of CABS, and a discussion of its limitations.

a. The Effectiveness of CABS

CABS is an extremely useful system in the performance evaluation of MBDS. A complete set of performance measurement tools are generated with minimal user intervention. Numerous parameters are adjusted to insure database-independence and application-independence when generating the benchmarking information. It provides the user with a systematic method to generate the test databases and the test transaction mixes, to collect the test results, tc interpret the results, and to verify the results against established measures (benchmarks).

b. An Overview of CABS

The primary operation of CABS is the generation of test transactions and test databases that may be used for the benchmarking of parallel, multiple-backend computers, in particular, MBDS. A design feature of the system is to minimize the required input from the user. The user needs only to supply three essential elements of information to CABS:

1) the number of backends in the system to be tested,

2) the amount of disk storage per backend, and

3) the size of the data transfer from secondary storage (disk) to primary storage (main memory).

Once the user has provided the necessary information, CABS automatically generates the test databases and the test transaction mixes. It also provides a comprehensive report to guide the user through the testing process. The CABS system is able to generate test sets for almost any combination of input data.

For a given test database, two sets of configurations are generated, one for the measurement of the response-time reduction, and the second for measurement of the response-time invariance. The number of configurations within each set is dependent on the number of backends in the system. CABS determines the correct database size to evenly distribute the data over all backends, so the performance-gain claims may be verified. To verify growth-capacity cıaims, the database must incrementally increase in size, while increasing the number of backends. The total number of configurations required is (2M - 1), where M is the number of backends in the system. [Ref. 12]

c. CABS Limitations

CABS contains several imbedded assumptions about the size of the test database. When performing initial performance evaluations, it may be desireable to utilize relatively small databases, allowing preliminary verification of performance claims. Once the claims are initially verified, a comprehensive performance evaluation may be conducted. This would preclude the need to load megabytes worth of data if the system did not initially meet expectations. CABS does not allow for such an initial verification. Because of internal calculations and size dependencies, CABS generates a minimum database size of four megabytes per backend.

The second CABS limitation is a minor one, and may easily be fixed by someone requiring the use of CABS. CABS does not include a target-list in the transaction mixes. The requests generated include the required qualifications so the correct amount of data is accessed and retrieved for the request, but omits a target-list. The user must either modify the CABS software directly, or must edit each individual test transaction mix file and add the appropriate target-lists.

B. SYSTEM CONFIGURATION CONSIDERATIONS

This section contains a detailed discussion of the relationships between critical internal constants within the MBDS software, and serves as an aid to those evaluating the performance of MBDS in the future. A change to any of the constants discussed in this section results in an entirely new instantiation of the database system software. A change in one constant many times necessitates the modification of other interrelated constants. It is the evaluators responsibility to carefully consider each change. The evaluator must reconfigure the database system so it can handle the demands placed upon it during the evaluation. The configuration must also accurately reflect a database system which may be used in real-world applications, so that the results of the evaluation reflect the actual performance of the system.

In the remainder of this section, we first provide an example demonstrating the relationships between physical size parameters. Next, we present an example illustrating the determination of system message-passing constants. Finally, we describe other system constants, and explain the relationships between them. The reader should be familiar with the tables in each of the sections, which contain a list and brief description of the critical system constants to be discussed.

1. Physical Size Relationships

This section discusses, by way of example, the interrelationships between internal parameters in the MBDS software. The internal parameters or constants are as follows:

(a) RecSize: This is the maximum size of each physical record. The system has only one record size defined.

(b) ANLength: The length of an attribute-name.

(c) AVLength: The length of an attribute-value.

(d) TrackSize: This is the size of a logical track.

(e) MAX_ADDRS: The number of physical addresses required. Defined by the number of records in the database, the record size, and the track size.

The record size (a) is normally the parameter that is adjusted most often to correspond to the different test configurations. For performance evaluation, we can assume that there is no record body, only attribute-value pairs (keywords). This assumption is justified by the fact that the retrievals made during the performance evaluation are for keywords only. As an example, we choose a record size of 200. (Size and length remain dimensionless in this discussion - actual internal representation is machine dependent.) This suggests that the size of the attribute-value pair be defined in terms of the record size of 200. The attribute-name (b) and attribute-value (c) parameters (lengths) are the smallest units of length in the parameter definitions. If we choose an attribute-name and attribute-value length of 10 each, this gives us a combined length of 20 for a keyword. This allows a total of 10 attribute-value pairs for each record.

MBDS currently uses the notion of a logical track size to store data on secondary storage. This requires the logical track size (d) to be defined. It is not possible to extend the storage of a record across a track boundary. This necessitates the determination of track size in terms of record size. In addition, each record requires a small amount of storage overhead (approximately 4 bytes per record) which must be accounted for in the track size. For our example, we choose a track size of 1024, providing storage for 5 records in each track. These parameters define the physical structure of the database.

The directory-management process in each backend is responsible for the generation of addresses for accessing the clustered records in the database. The number of addresses for each backend is finite, and is determined in relation to the track size and the database size. The size of the database is predetermined, and is based on the size required for the evaluation. The total number of records in the database divided by the number of records per track yields the total number of required tracks for the database. This number is the maximum number of addresses (e) required, and must be set accordingly.

2. Message-passing Constants and Relationships

This section discusses the relationships between the size of the internal message buffers, and the size of the actual messages. The constants are as follows:

(a) ResBufSize: The length of the result buffer in record processing.
(b) PP_ResBufSize: The length of the result buffer in post processing.
(c) RESLength: The length of the result buffer in the test interface of the controller.
(d) MSGLEN: The length of the messages between processes and computers.

An important factor is the size of the buffers and messages within MBDS. There are buffer sizes defined in record processing (a), post processing (b), and the test interface (c). All of these buffer sizes should be the same to avoid any problems when transferring data between buffers in different processes. The transfer of data from the disk I/O process to record processing in the backends require the message-buffer (d) sizes to match or exceed the track size. This allows record processing to extract an entire track and only require one buffer. For our example, we choose a track size of 1024. The size of the messages (d) passed between processes and computers must be large enough to hold at least one buffer and the associated message overhead. The overhead includes information such as the traffic-id, the request number, the message number, and delimiting information within the message itself, such as beginning-of-message, beginning-of-result, end-of-result and end-of-message. This overhead normally amounts to approximately 10-20 bytes. Accordingly, for our example, we choose a message size of 1040.

3. Other System Constants

There are several other parameters that should also be considered when configuring MBDS. These constants, with brief descriptions, are listed below.

(a) REQLength: The maximum length for any request.

(b) TILength: The number of digits in a traffic unit id. Defines the maximum number of transactions that may occur per session.

(c) RT_MAX_ENTRY: The maximum length of the internal request table for storing the parsed request.

(d) QR_MAX_ENTRIES: The maximum length of the query table. Should be the same as (c).

(e) MAX_FIELDS: The maximum number of attribute-value pairs for a database file or template.

(f) MaxCids: The maximum number of clusters allowed in a database.

(g) QR_MAX_DIDS: The maximum number of Descriptor-Ids found for any query.

(h) ReqMaxDidSets: The maximum number of DID sets for any one request.

(i) DT_MAX_DESC: The maximum number of descriptors in the internal descriptor table.

The request length (a), is the maximum allowable length in terms of the number of alphanumeric characters a request may be. The traffic-unit id (b) is simply a sequential numbering of all traffic-units issued throughout a session with the database system. If the constant specifies, for example, five digits in each id, then there could be a maximum of 10^5 possible ids. which is normally sufficient.

The request table constants, (c) and (d), are related to the number of fields allowed (e) in a record template or file. A field is defined as an attribute-value pair. The request table has an amount of overhead that is dependent on the type of request. For example, an Insert request has seven entries in the table used by the system (see Chapters IV and V). If the number of fields were 30, there would be a requirement for 60 additional entries in the table to accommodate the attribute-name and attribute-value defined for each field. Therefore, the size of the table would have to be, as a minimum, 67.

The maximum number of clusters permitted in the system is described by the constant in (f). The number of clusters required is dependent on the descriptors for a database. Each directory attribute must have associated with it a collection of descriptors, located in the DDIT (see

Chapter II.C.). The characteristics of the user data, along with the collection of descriptors for each directory attribute, together define the number of clusters required.

The entries (g) through (i) are all related to the characteristics of the data defined in the descriptor-ids for each directory attribute. The constant in (g) simply defines the number of allowable descriptor-ids for any given request. The constant in (h) is related, in that it defines the maximum number of allowable descriptor-id sets for any given request. The size of the set is determined by the cross-product of all descriptors accessed for any one request. The constant in (i) is also similar because it defines the limitation on the number of descriptors allowed for a database. These constants do not necessarily have a specific value that can be set to guarantee reliable operation of MBDS. Each of the values depend on the data to be used in the system. Thus, the evaluator must know what data is to be used in the evaluation in advance, and configure the system accordingly.

Lastly, we collect and present the constant values discussed in this section in Figure 3.a. The values marked with an asterisk in Figure 3.a indicate values that are dependent on the test database used. The values listed were determined for a database of up to 2500 records and 25 descriptors.

Parameter	Value
RecSize	200
ANLength	10
AVLength	10
TrackSize	1024
MAX_ADDRS	500*
ResBufSize	1024
PP_ResBufSize	1024
RESLength	1024
MSGLEN	1040
REQLength	500*
TILength	5*
RT_MAX_ENTRY	67
QR_MAX_ENTRIES	67
MAX_FIELDS	30
MaxCids	25*
QR_MAX_DIDS	20*
ReqMaxDidSets	25*
DT_MAX_DESC	25*

Figure 3.a Parameter Values

IV. AGGREGATE RETRIEVAL

In this chapter we discuss the aggregate retrieval operation, and the implementation and integration of the operation into the Multi-Backend Database System, MBDS. The problems associated with implementation, integration, and testing are also discussed. The execution of an aggregate request is traced in detail from both a user's perspective and at the system level.

The first section of this chapter deals with some of the requirements that are involved with the implementation of the new aggregate operation. A description of the operation is given, including example requests and corresponding results. The second section describes the actual implementation of the aggregate retrieval and provides some insight into the problems encountered with the integration into the MBDS system. Finally, a discussion of applicable testing methodologies and the results obtained by testing the newly developed code are given.

A. REQUIREMENTS

A modern database should be capable of performing more than the basic insert and retrieval operations. It is important that the user be able to retrieve data and have it presented in a meaningful way. The aggregate retrieval operation provides the user with just such capability by applying statistical functions on the data that is accessed and retrieved. An operation such as obtaining the count of a specific data element can provide extremely useful data which is not inherently available in the data itself. The most common numerical functions have been included in MBDS. The included aggregate operators are sum, minimum, maximum, count, and average (mean). As the reader may recall, a standard retrieve request consists of a query and a target-list. The query specifies which records are to be retrieved. The target-list specifies which attributes are to be retrieved and returned in the response. The aggregate operator is considered an optional part of the primary retrieval operation, and is capable of being applied to one or more of the attributes contained within the target-list.

1. The Design of the Aggregate Retrieval

In this section we discuss the general algorithm for processing aggregate operations in MBDS. A short review of how the aggregate operators are processed is also presented. All of of the operators perform in the same general fashion with most of the work being performed in the backend. Each backend performs the aggregation for the data present on the particular backend. The results of the aggregation are sent to the controller for further aggregation. When results have been received from all the backends, the operation is completed and the results are sent to the user via the test interface process.

With the exception of average, the processing of the different aggregate operators is very similar. For count, as each record is retrieved in a backend, the value for the requested attribute is counted. In post processing, the count from each of the backends is added to obtain the overall count. For maximum, as each record is retrieved the attribute value is compared with a stored value (initially a very small number) and if it is greater than the stored number it replaces it. In post processing the maximum value from each of the backends is compared and the final value is obtained. The minimum works in the same way except it replaces the stored value only if it is less than the stored value and it is initalized with a very large number. The operation of minimum in post processing is also similar. Sum works by maintaining a running total of each value as it is retrieved in the backend. In post processing the individual sums from the backends are added to produce the final value to be sent to the test interface. The average operation is somewhat different from the others. For average, each backend keeps, two pieces of data, a count and a sum for the requested attribute value. After the last record has been retrieved in a backend, the sum and count is sent to post processing. In post processing, the individual sums and counts are added. After results have been received from all backends the average value is computed and sent to the test interface.

2. Example Requests and Results

We now present an overview of the use of the aggregate operators and the corresponding results produced. The syntax of the aggregate retrieval operation is similar to that of a normal retrieval. When using an aggregate operator, the operator is specified in the target-list with the attribute it is to operate on. As an example the retrieve shown below returns all SNO and PNO attribute-values, as well as the average of all QTY attribute-values.

```
[RETRIEVE(File=Ship)(SNO,PNO,AVG(QTY))]
```

When the request is executed, the aggregate operator is applied to all values for the attribute QTY in the database.

```
(<SNO,                S6>,              <PNO,              P2>)
(<SNO,                S2>,              <PNO,              P1>)
(<SNO,                S3>,              <PNO,              P3>)
(<SNO,                S5>,              <PNO,              P6>)
(<SNO,                S2>,              <PNO,              P4>)
(<SNO,                S4>,              <PNO,              P7>)
(<SNO,                S7>,              <PNO,              P9>)
(<SNO,                S5>,              <PNO,              P8>)
(<SNO,                S7>,              <PNO,              P3>)
(<SNO,                S2>,              <PNO,              P1>)
(<SNO,                S1>,              <PNO,              P2>)
(<SNO,                S3>,              <PNO,              P2>)
(<AVG(QTY), 1438.417>)
```

We note that the results of the retrieval are presented as attribute-value pairs. The aggregate results are listed at the end of the result list also as an attribute-value pair.

Following are some additional sample retrievals for all of the newly implemented aggregate operators in MBDS.

```
[RETRIEVE(File=Ship)(SNO,PNO,MAX(QTY))]
[RETRIEVE(File=Ship)(SNO,PNO,MIN(QTY))]
[RETRIEVE(File=Ship)(SNO,PNO,SUM(QTY))]
[RETRIEVE(File=Ship)(SNO,SUM(QTY),AVG(QTY),COUNT(PNO))]
```

The aggregate operators may be applied in any combination on the same or different attributes. However, the operators min, max, sum, and average may be applied only to attributes specified to have numeric attribute values. The count operator may be applied to attributes having numeric or non-numeric attribute values.

B. IMPLEMENTATION AND INTEGRATION

This section describes the process of implementation and integration of the aggregate operators into the existing MBDS software. Our primary goal was to implement the functions in such a manner as to allow the functions to maximize the work done by the backends, and to minimize the work done by the controller. This goal was in step with the original goals of MBDS. Another implementation goal was to utilize the functions in the existing system to the greatest extent possible with minimal modification.

Our goals required that we perform a comprehensive study of the existing MBDS software as well as the many supporting technical reports. Because the need for aggregate operators had been foreseen in the initial design of MBDS, our goal was made much more realizable. The tasks of parsing, syntax checking, and formatting the request table had already been implemented in the request preparation process of the controller. Several functions had been stubbed in the source code in both the record processing and the post processing processes. The implementation was divided into two main areas; record processing in the backend and post processing in the controller, thus evenly distributing the effort.

1. The Basic Operation

The operation of a retrieval with aggregate operators is very similar to the operation of a retrieval with no aggregate operators. When a retrieve request arrives in record processing, the target list, as shown in Figure 4.a, is searched for any aggregate operators.

```
0  Beginning of Request
1  Traffic Id
2  Request Number
3  Routing Indicator Code
4  Request Type Code
5  Number of Keywords or Predicates
   Value
     .
     .
   End of Conjunction
   End of Query
   Attribute
   Aggregate Opcode or 000
   Attribute
   Aggregate Opcode or 000
   Attribute
     .
     .
   End Of Target List
   Attribute for By-Clause
   With
   End of Request
```

Figure 4.a Target List for a Retrieve

If an aggregate operator is found, a structure is allocated for use in processing the request. As each record is retrieved, the aggregate operation is performed for the specified attribute, and the rest of the record is buffered in preparation for sending to post processing. When the last record for a backend is read, the aggregate result, along with any other remaining data, is sent to post processing.

2. Execution of the Aggregate Request

This section describes the sequence of events in the execution of a retrieve operation which includes an aggregate operator. The reader will recall that MBDS uses intra- and inter-computer messages for control and transfer ring data. Recall that in Figure 2.g we have listed the types of messages used by MBDS for internal process coordination and control. Figure 4.b schematically displays the controller and backend processes as well as the messages which are sent between the individual processes and between the backend and controller for a retrieval with an aggregate operator. The order in which the messages are passed are denoted alphabetically (e.g. 'A' is first). The number following the letter denotes the message type as listed in Figure 2.g.

A retrieve request with an aggregate operator originates in the Test Interface process. The completed request is sent from the test interface to the request

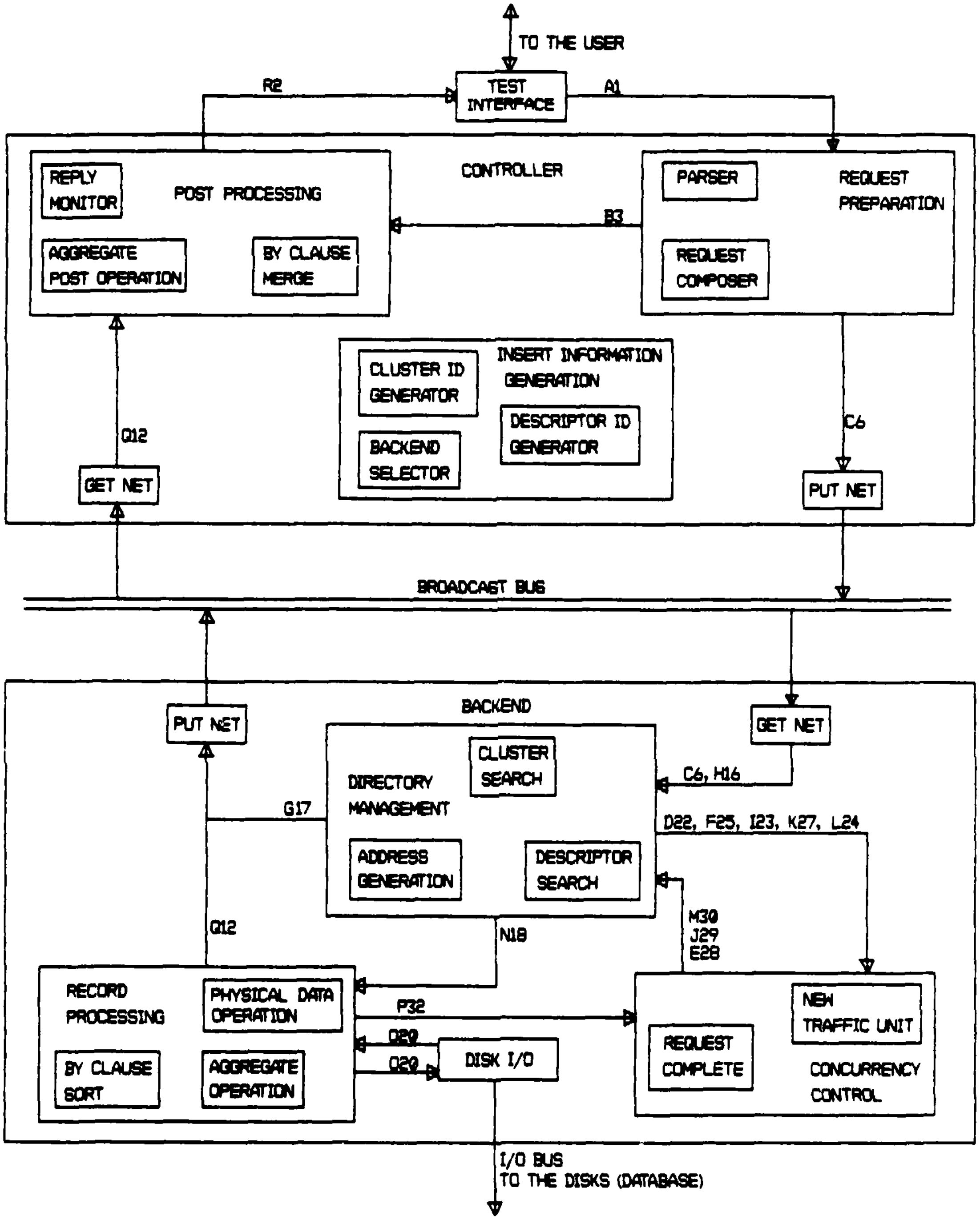


Figure 4.b The Sequence of Messages For Executing a Retrieve With Aggregate Operator

preparation process for parsing, syntax checking, and formatting into a request table (A1). Request preparation notifies post processing of the number of the requests in the transaction (B3). Upon completion of this, request preparation sends the parsed traffic unit to directory management (C6). Directory management calls on concurrency control to lock the directory attributes (D22). After the attributes are locked, concurrency control notifies directory management of the event (E28), and directory management begins a descriptor search for the retrieve. Once this is completed, directory management notifies concurrency control to release the locks on the attributes (F25), and directory management broadcasts the descriptor ids to the other backends (G17). The directory management processes in the other backends are also sending their descriptor ids to the directory management in this backend (H17). The backends use the information received from all of the other backends to form descriptor-id groups. These groups are are sent to concurrency control to be locked (I23). After concurrency control notifies directory management that these groups are locked (J29), directory management performs a cluster search and notifies concurrency control to release the locks for the retrieve (K27). Next, directory management sends the cluster ids for the retrieval to concurrency control (L24). Concurrency control notifies directory management when the clusters have been locked (M30). At this time, directory management determines the disk addresses for the request. Directory management then sends the retrieve request and its disk addresses to record processing (N18). As needed, record processing interacts with disk I/O for database information (O20). When record processing finishes executing the retrieve, concurrency control is notified (P32) that the request is done, and the locks on the cluster ids are released.

The aggregation for each of the operators besides the average operator, is performed in the backend. For the average operator, the data values are counted and a running total kept, in each backend. When a request is completed in the backends, the sums and counts from each backend are sent to the controller. These values are added and the average is then calculated in post processing. After the retrieval results have been aggregated in a backend, the results are sent to post processing (Q12) for further aggregation with the results from other backends. When the results from all of the backends are received, the aggregate operation is completed and the results sent to the test interface for display (R2).

C. TESTING

In this section we discuss one of the most important and time consuming stages in the software life cycle, the testing of software. The objective of testing is to locate and correct as many errors as possible. The testing of the retrieve with aggregate operations was done with the objective of revealing specific classes of errors.

The testing process took place in two phases. The first phase occurred prior to the integration of the new software into MBDS and the second phase after integration. Several techniques were used in testing the aggregate retrieval. These techniques include boundary testing, unit testing and structure testing. Several special cases were also considered, including testing all operators on the same attribute-value pair, and retrieval from a backend which contained no data matching the query. After integration, testing was initially performed using a single backend, and was followed using multiple backends. Overall, our testing process followed the aggregate request through it's execution path as shown in Figure 4.b. In the the process, we are confident that our testing has been rather complete and comprehensive.

The first phase involved unit testing of the record processing and post processing modules prior to integration into MBDS. Unit testing of the individual modules was accomplished utilizing a test harness written specifically to test each module. With a few slight modifications to parameters, the aggregate retrieval module was then used to drive the testing of post processing. It was at this time that testing of boundary conditions, and testing of particular program paths for each module occurred. No significant errors were discovered in this phase.

The second phase of testing involved the testing of the aggregate modules after their integration into MBDS. This proved to be far more difficult than the first phase of testing. MBDS is a large system, (approximately 35000 lines of code) resulting in each cycle of the compile, link, test, and debug loop to be very time consuming. Because of the size of MBDS this phase of testing was divided into three parts. The first part was the testing of the functions in record processing. During this part, a minor error was discovered which was eventually traced to the parser. A retrieval operation with a count aggregate operator was arriving in record processing as a sum operator. This error, as well as other minor errors, were detected and easily corrected during this stage. Part two involved testing of the functions in post processing. Errors which were found were relatively minor. The third part consisted of repeating the testing performed in parts one, and two using multiple backends. Very few errors were discovered and those that were found were easily corrected.

V. SORTED RETRIEVALS

In this chapter we discuss the need and functionality of the sorted retrieval operation. First we describe the motivation behind the implementation and integration of the sorted retrieval into MBDS. Included are some example requests and the corresponding responses. Next, we examine the use of the sorted retrieval, and it's operation at the software system level in detail, and include a detailed trace of a so-ted retrieval request. Third, we discuss the testing of the sorted retrieval operation or by-clause and mention some of the difficulties encountered. Finally, we discuss the combination of the aggregate and the by-clause operations. The integration of the two independent functions as well as the ensuing problems are discussed.

A. REQUIREMENTS

A modern database should be capable of performing more than just the basic insert and retrieval operations. Data can take on more meaning when the user can issue a retrieval request, and have the resulting data displayed in a sorted fashion. The by-clause provides the user with just such a capability. This sorted retrieval, or by-clause, is an optional part of the retrieval operation and can be applied to both string and numeric data as a sorting attribute.

1. The Design of The Sorted Retrieval

The overall design of the sorted retrieval is straightforward. The data local to each backend is hashed and stored in the virtual memory. After the last record for a by-clause is read in a backend the hashed records are sorted. The sorted records are then sent to the controller for merging. The merged records are then sent to the user via the test interface process.

When a request with a by clause is received by a backend a hash table is allocated for the request. As each record is retrieved it is hashed on the sorting attribute, it's address is stored in a hash table and the record is stored in the virtual memory. When the last record in a retrieve is read, the buckets are sorted and sent to post processing for merging. When the records arrive in post processing, the records for all backends are merged and the results sent to the user via the test interface process. In the design of the sorted retrieval, the retrieve-common implementation [Ref. 14] was studied closely. The hashing algorithm and hashing structures designs utilized in the retrieve-common implementation were used in the design of the sorted retrieval. The reader is referred to [Ref 14] for a more detailed discussion of these algorithms.

2. Example Requests and Results

In this section we provide the reader with a brief overview of the operation of the sorted retrieval at the user level. To utilize the sorting function of the retrieval operation, the user must specify which attribute in the target-list is to be used as a sorting attribute. The retrieval request may use the optional by-clause on any attribute for the particular database, as long as that attribute is within the requests target-list. The sorting is done in relation to the user specified attribute, and always responds with the data sorted in ascending order. Below is a sample retrieval using the optional by-clause, followed by [RETRIEVE(TEMP=Part)(PNO,NAME,CITY)BY CITY] the results of the retrieval.

```
(<PNO, P2>, <NAME, Washer>, <CITY, Carmel>)
(<PNO, P1>, <NAME, Nuts>, <CITY, Columbus>)
(<PNO, P3>, <NAME, Staples>, <CITY, Gilroy>)
(<PNO, P5>, <NAME, Bolts>, <CITY, London>)
(<PNO, P9), <NAME, Screw>, <CITY, Monterey>)
(<PNO, P7>, <NAME, Nails>, <CITY, Salinas>)
```

We note that the results of the retrieval are presented as attribute-value pairs sorted in ascending order by city.

B. IMPLEMENTATION AND INTEGRATION

In this section we describe the implementation, and integration of the optional sorted retrieval operation into the existing MBDS software. First, the basic operation of the sorted retrieval at the software level is presented. Then, the basic functionality of the process is discussed. This is followed by a detailed trace of an example sorted retrieval.

Our goals for this implementation were the same as for the aggregate operator implementation. Like the aggregate operator implementation, the need for the by-clause had been foreseen in the initial design of MBDS. The tasks of parsing, syntax checking, and formatting the request table had already been implemented in the request preparation process of the controller. Thus our implementation was divided into two areas, record processing in the backend and post processing in the controller.

1. The Basic Operation

The operation of a sorted retrieval in the backend is nearly identical to a non-sorted retrieval. When a retrieve request arrives in record processing, a search is performed on the target list, as shown in Figure 4.a, to see if it is a sorted retrieval operation. If it is a sorted retrieval, a hashing table is allocated for use in processing the response data. Each record that matches the query for the request is hashed on the sort attribute-value into the virtual memory, and the hashed address is stored in the hash table. When the last record has been retrieved and hashed, the data is prepared for sending to post processing.

First, each table entry is checked for primary buckets. If there is a primary bucket a check for overflow buckets is made. Every bucket(s) is sorted and then sent to post processing. As each message arrives in post processing the buckets are extracted and merged into a hash table. The hash table used is the same size (i.e., it has the same number of index entries) as the table used in record processing. Therefore, the bucket number, along with the corresponding data, is sent in the message. When the message arrives, there is no need to perform the hashing calculations because the bucket number is already known. When the results from the last backend have arrived and been merged, the data is taken from the hash table and sent to the test interface process. The data is then simply displayed as ordinary data.

2. Execution of the Sorted Retrieval

This section describes the sequence of events in the execution of a retrieve operation which includes a by-clause. The reader will recall that MBDS uses intra- and inter-computer messages for the control and transferring of data. Figure 2.g lists the types of messages used by MBDS for internal process coordination and control. Figure 5.a schematically displays the controller and backend processes as well as the messages which are sent between the individual processes and between the backend and controller for a retrieval with an optional sort. The order in which the messages are passed are denoted alphabetically (e.g. 'A' is first). The number following the letter denotes the message type as listed in Figure 2.g.

A retrieve request with an optional sort originates in the test interface process. The completed request is

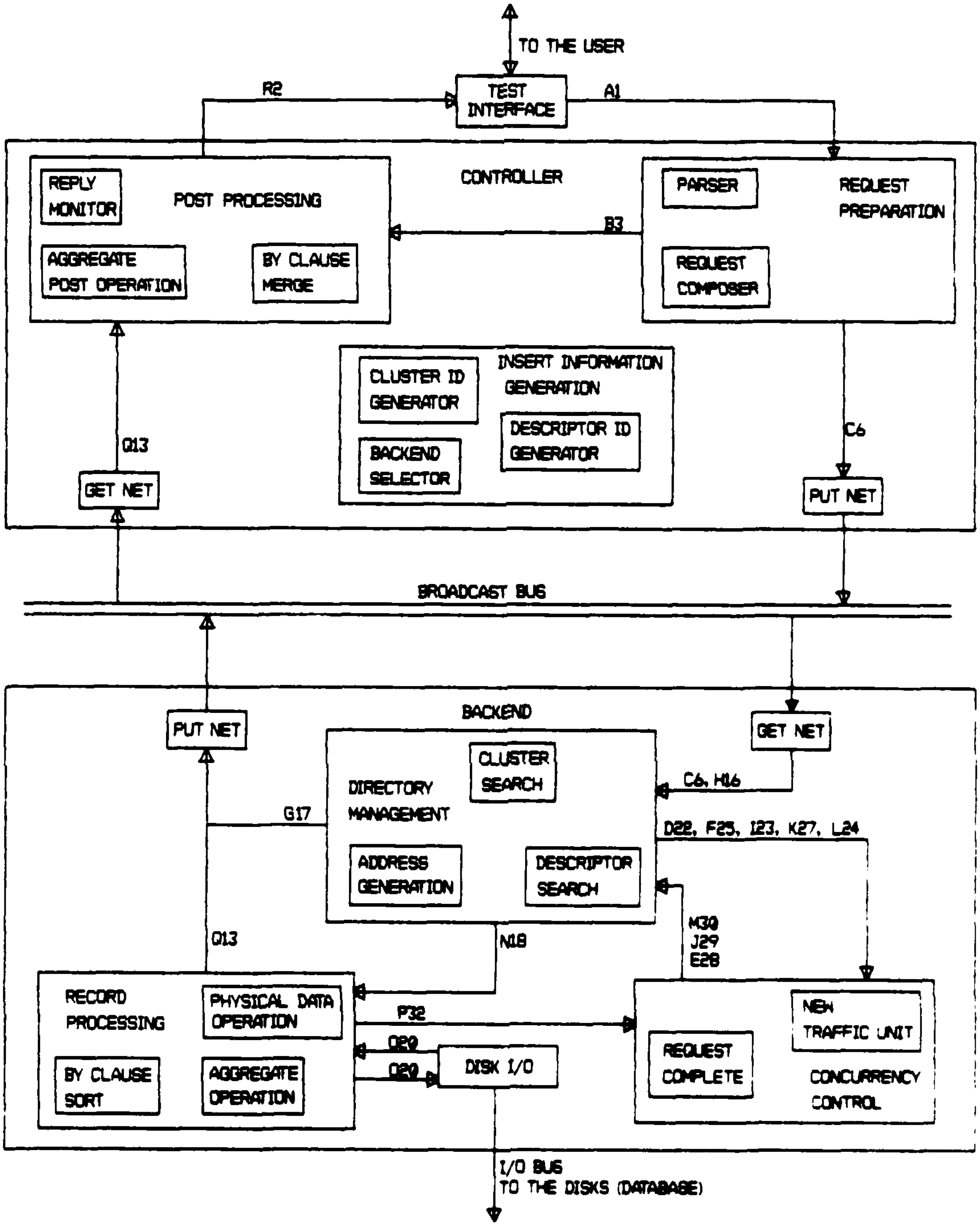


Figure 5.a The Sequence of Messages For Executing a Retrieve With Aggregate Operator

sent from the test interface to the request preparation process for parsing, syntax checking, and formatting into a request table (A1). Request preparation notifies post processing of the number of the requests in the transaction (B3). Upon completion, request preparation sends the parsed traffic unit to directory management (C6). Directory management notifies concurrency control to lock the directory attributes (D22). After the attributes are locked, concurrency control notifies directory management of the event (E28), and directory management begins a descriptor search for the retrieve. Once this is completed, directory management notifies concurrency control to release the locks on the attributes (F25), and directory management broadcasts the descriptor ids to the other backends (G17). The directory management processes in the other backends are also sending their descriptor ids to the directory management in this backend (H17). The backends use the information received from all of the other backends to form descriptor-id groups. These groups are sent to concurrency control to be locked (I23). After concurrency control notifies directory management that these groups are locked, (J29) directory management performs a cluster search and notifies concurrency control to release the locks for the retrieve (K27). Next, directory management sends the cluster ids for the retrieval to concurrency control (L24). Concurrency control notifies directory management when the clusters have been locked (M30). At this time, directory management determines the disk addresses for the request. Directory management sends the retrieve request and the corresponding disk addresses to record processing (N18). As needed, record processing interacts with disk I/O for database information (O20). When record processing finishes executing the retrieve, concurrency control is notified (P32) that the request is done, and the locks on the cluster ids are released.

After the last record is retrieved for a backend, the individual buckets are sorted and then sent to post processing (Q13). As the records arrive in post processing they are merged with the records already received from other backends. After the results have been received from all the backends the results are sent to the test interface for display (R2).

C. TESTING

In this section we discuss the testing of the sorted aggregate retrieval. Testing was divided into two primary phases. The first phase occurred prior to the integration of the new software into MBDS and the second phase after integration. The testing methods used were the same as those used for testing the aggregate retrieval software (see section 4.C).

For the first phase of testing the software was compiled and tested independently of MBDS. A test harness was used to test selected paths in the modules for normal as well as boundary conditions. During this stage a symbolic debugger was used to assist in debugging. Many minor bugs were found and corrected during this phase.

The second phase of testing involved testing of the new modules after integration into MBDS. Because of the size and complexity of MBDS this phase of testing was divided into three parts. The first part involved testing of just the modules in record processing using a single backend. The second part involved the testing of the record processing modules and the post processing modules, again using only a single backend. The third part of testing consisted of repeating all earlier testing using multiple backends. No major errors were discovered in this phase. Those that were found were in general easily corrected. The testing of the combination of aggregate retrieval with the by clause is discussed in Section 5.D.

D. THE COMBINATION SORTED AND AGGREGATE RETRIEVAL

This section contains a discussion on the integration of the two retrieval operations, aggregate retrieval and sorted retrieval. Each operator was implemented and integrated into the existing MBDS software independently. The goal was to ensure the combined operation of the two operators.

The two operators, aggregate retrieval and sorted retrieval, each intercept data in record processing. The data intercepted has already been extracted from the database and is ready for sending to the controller. The data is taken by the new operations, and acted on accordingly, i.e. hashed if it is a sorted retrieve, or stored in the aopropriate data structure if it is an aggregate operation. Both of the operations intercept the data at approximately the same place in the code. In order for the two operations to act on data retrieved for one request, they each must be provided the necessary data.

The data required by each of the two operations is disjoint, i.e., the sorted retrieval operation doesn't require any of the data that the aggregate operation uses. If a retrieval has a by-clause, and aggregate operation on the same attribute then that attribute-value pair will be retrieved twice for each record. This allowed us to give one of the operations precedence over the other, so data not used by the first operation is passed on to the next operation. The aggregate operation was initially designed to take only the data it required, then passing the rest of the data on to post processing. We simply altered the aggregate operation function so that it passed along the data to the sorted retrieval operation if the request contained a by-clause. The data received by the sorted retrieval operation is the same data that would have been received if there was no aggregation being performed. This made the combination of aggregate and sorted retrieval transparent to the sorted retrieval operation, and required only minor changes to the processing for aggregate operations. Each of the operations sends its data back to post processing at the end of a request using their respective message types.

The arrival of two separate messages in post processing for a single request posed another problem to be solved. Under normal circumstances, there may be any number of messages arriving in post processing in response to a request. The last message for a particular request is labeled with a special character in the message signifying 'End-of Request',i.e., the last set of results for a request from a particular backend. When both operations sent independent messages to post processing for a single request, post processing had to be able to know that a combination request was being performed. Once this was known, post processing would wait for two 'End-of-Result' messages instead of the usual one.

A third obstacle arose at the end of a combination retrieval. When acting independently, the two operations each released memory and sent the response data to the test interface at the end of the request. When a combination request was made, the first operation to receive and process all of its data sent the data to the test interface and released the memory for the entire request. This had to be modified so that the memory was released only after both operations had sent their results off to the test interface process.

VI. CONCLUSIONS

In this final chapter, we provide some concluding remarks. In the first part of the chapter, we furnish a summary of the work conducted for this thesis. Next, we discuss the difficulties and problems encountered while completing the work for this thesis as well as addressing some of the solutions that were attempted. Finally, we provide some recommendations for future efforts.

A. SUMMARY

Performance problems and upgrade issues have always been an obstacle in traditional mainframe-based systems and software single-backend systems. The Multi-Backend Database System, or MBDS, utilizing the software multiple-backend approach, attempts to overcome these problems through specialization of the database operations on multiple, dedicated backends. The two goals of MBDS are to overcome upgrade and performance problems normally associated with traditional systems. In this thesis, we have provided a methodology for evaluating MBDS in terms of response-time reduction and response-time invariance. We discuss the utilization of CABS in the benchmarking process. We find that CABS is most useful when conducting benchmark tests using very large test databases. We then discuss the MBDS internal parameters and constants, and consider their importance in the configuration of MBDS.

This thesis also presents our design, implementation, and integration work on two new types of retrieval operations, the aggregate retrieval and the sorted retrieval. The aggregate retrieval operation provides the user with an integral tool for the interpretation of data. The aggregate retrieval operation allows the user to retrieve data, and perform any number of the five primary aggregate operators, COUNT, SUM, MIN, MAX, or AVERAGE, on that data. The sorted retrieval operation provides the user with a means of organizing data in a way meaningful and useful to the user. The user may retrieve data from the database, and have the data presented in a sorted manner in relation to any of the attributes. A discussion of both of these operations and an example of its use has been provided.

B. DIFFICULTIES ENCOUNTERED

There were several problems encountered during the course of the thesis work. This section explains the problems as well as the solutions that were attempted.

1. Message Passing

The primary problem encountered was the message passing facilities provided for inter-computer messages in MBDS. The actual benchmarking of MBDS was not performed due to this problem. MBDS functioned with no difficulties on the previous hardware and software configuration. This configuration was similar to the current configuration, but utilized a Motorola 68010 CPU. When MBDS was moved to the current configuration, utilizing the faster Motorola 68020 CPU, difficulties began to arise. It was found that messages were being sent, but not being received at their destinations. The messages were simply being lost. Since MBDS performed flawlessly before the new configuration, the possibility of the MBDS code as causing the problem was ruled out. It seemed unlikely that the new CPU itself caused the problem, but the faster execution speed of the new CPU could have contributed. The problem was found to lie primarily with the transmission of messages between computers in MBDS, i.e., among the backends and the controller. Since MBDS depends on the reliable transmission of broadcast messages, this made for a very distressing situation. Messages are broadcast from the controller to backends, and between backends. The Unix operating system provides the underlying protocols used for the message passing. Unfortunately, the only available protocol for broadcasting in the current operating system is an unreliable protocol (which, we note, was more reliable in the 68010 environment). In order to perform the benchmarking of MBDS, we investigated a number of solutions to our problem.

The first solution involved trying a new version of the Unix operating system. The main difference between the old and new versions was the inclusion of multiple software buffers for incoming messages. The older version provided a single buffer, while the new version provided up to eight buffers. It was thought that a possible reason for losing inter-computer messages, was that the messages were being received too fast. With only one buffer, if a message arrived before its predecessor was processed, the buffer was still full, and the new message was lost. However, the installation of the new operating system made no difference.

The second solution was to use newer, faster Ethernet controller hardware. It was thought that if the hardware could process the messages faster, there would be less of a chance of the software buffers being full, and hence lost messages. The new Ethernet controller cards were rated to be 40 percent faster than the existing controller cards. This solution caused messages to be passed faster, but had no effect in terms of lost messages.

Another factor to be considered is the usage of the network being used by MBDS. It was thought that the network was possibly being overworked at times of peak broadcast activity, and was therefore unable to handle all of the messages. This possibility lead us to the monitoring of the Ethernet activity. It was found that the network was not being overworked at all. At times of peak broadcasting, the network rose to a peak utilization of under five percent (for four backends). This certainly was not the cause of the problem.

The last attempted solution concerned the amount of primary memory in each of the backends. Through monitoring of the operating system processes, it was found that MBDS processes were being paged to disk rather often. The paging was due to the size of the memory in the backends. Each of the backends contains two Mbytes of main memory. The amount of available memory is approximately 820 Kbytes. This relatively small memory space caused the swapping of the MBDS processes in the backends. The paging sometimes occurred at crucial times when the process which processes the message is paged out, possibly causing the lose of a message. An obvious solution to this problem was to make the MBDS processes responsible for handling network message traffic (i.e., get-net and put-net in the backends and the controller) to be memory resident (i.e., incapable of being paged out). This solution was tried, but did not solve the problem.

2. System Size

The size of MBDS and the Unix operating system both contributed to a very steep learning curve for students working on the MBDS system. The amount of information initially required by a student to work on MBDS is very large, and requires a substantial portion of the students alloted thesis time. Besides learning an entire operating system, the student must learn both the concepts and implementation details of a very large database system. In order to understand the MBDS implementation, the student must also become very proficient in the C language. These requirements constrain the students time to work with the system.

The system utility, Make, also caused some problems. Makefiles are used for defining dependencies between files in a system. The Make utility uses the dependencies to create an up-to-date version of the system, by compiling only those files which have been changed since the last compilation. The utility is meant to make the management of a large system easier. The makefiles that have been defined for MBDS caused a lot of confusion. The makefiles are written with very intricate and complicated dependencies. Additionally, there are several versions of MBDS in use at any one time, and each version uses source code from other versions. The difficulty in modifying the makefiles coupled with the multiple versions made even the simple task of compiling the system very difficult at times. Another related problem arose, and has not yet been solved. When attempting to compile some portion of the system, the compiler would work for one version, but not for another. This would not normally be suspect, but in this case, both versions utilized the same source code and identical copies of the makefile. However, we do note that without the makefiles, our work on the system would have been severely impeded. Thus, it seems we can't live with them, and we can't live without them.

C. RECOMMENDATIONS FOR FUTURE EFFORTS

One of the foremost problems to be solved is the message passing problem. The system must be brought to a state where it can operate with any number of backends reliably. The most promising solution to the problem lies in the upcoming release of a new operating system supporting a reliable broadcasting protocol.

Once the problem with the message passing has been solved, the next obvious step is to benchmark MBDS. A complete and thorough test of the system must be conducted to further validate the MBDS claims of response-time reduction and response-time invariance. The methodology presented in this thesis coupled with the system configurations presented can be applied to the performance evaluation of MBDS to produce the required data to validate the claims of MBDS.

LIST OF REFERENCES

1. Hsiao, D. K., ed. *Advanced Database Machine Architectures*, (see preface), Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

2. Demurjian, S. A., et al., "A Multi-Backend Database System for Performance Gains, Capacity Growth, and Hardware Upgrade," *Proceedings of the 1986 2nd International Conference on Data Engineering*, 1986.

3. He, X., et al., "The Implementation of a Multi-Backend Database System (MBDS): Part II - The First Prototype MBDS and the Software Engineering Experience," in *Advanced Database Machine Architecture*, Hsiao (ed.), Prentice Hall, 1983.

4. Kerr, D. S., et al., "The Implementation of a Multi-Backend Database System (MBDS): Part I - Software Engineering Strategies and Efforts Towards a Prototype MBDS," in *Advanced Database Machine Architecture*, Hsiao (ed.), Prentice Hall, 1983.

5. Menon, M. J., "Design and Analysis of a Multi-Backend Database System for Performance Improvement, Functionality Expansion and Capacity Growth," Ph.D. Dissertation, The Ohio State University, Columbus, Ohio, 1980.

6. Naval Postgraduate School Report NPS52-87-018, *Towards a Better Understanding of Data Models Through the Multi-Lingual Database System*, by Steven A. Demurjian, May 1987.

7. Hsiao, D. K., and Harary, F., "A Formal System for Information Retrieval from Files," *Communications of the ACM*, Vol. 13, No. 2, February 1970; *Corrigenda*, Vol. 13, No. 4, April 1970.

8. Wong, E., and Chiang, T. C., "Canonical Structure in Attribute Based File Organization," *Communications of the ACM*, Vol. 14, No. 9, September 1971.

9. Rothnie, J. B. Jr., "Attribute-Based File Organization in a Paged Memory Environment," *Communications of the ACM*, Vol. 17, No. 2, February 1974.

10. The Ohio State University, Columbus, Ohio, Technical Report, OSU-CISRC-TR-77-7, *DBC Software Requirements for Supporting Relational Databases*, by J. Banerjee and D. K. Hsiao, November 1977.

11. Tung, H. L., *Design, Analysis, and Implementation of the Primary Operation, Retrieve-Common, of the Multi-Backend Database System (MBDS)*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1985.

12. Fenton, G. P., *A Computer Aided Design for the Generation of Test Transactions and Test Databases and for the Benchmarking of Parallel, Multiple Backend Database Systems*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1986.

13. Vincent, J. R., *A Performance Measurement Methodology for a Multi-Backend Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1985.

14. Hunt, A. L., *Implementation of the Primary Operation, Retrieve-Common, of the Multi-Backend Database System (MBDS)*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1986.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	6
4.	Computer Technologies Curricular Office Code 37 Naval Postgraduate School Monterey, California 93943-5000	1
5.	Professor David K. Hsiao, Code 52Hq Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	2
6.	Professor Steven A. Demurjian Computer Science and Engineering U - 155, Room 209 260 Glenbrook Road University of Connecticut Storrs, Connecticut 06268	2
7.	Lt. Frank E. Kelbe 720 J. Avenue Coronado, California 92118	3
8.	Lt. Dana S. Majors 32 El Verano Way Yuba City, California 95991	3

END

9-87

DTIC