A MULTI-BACKEND DATABASE SYSTEM FOR
PERFORMANCE GAINS, CAPACITY GROWTH AND HARDWARE UPGRADE *

Steven A. Demurjian and David K. Hsiao
Department of Computer Science
Naval Postgraduate School
Monterey, California 93943
U. S. A.

Jai Menon
IBM San Jose Research Laboratory
5600 Cottle Road
San Jose, California 95193
U. S. A.

ABSTRACT

Traditional database systems have long been plagued by performance problems when there is either an increase in the mainframe usage or in the database applications. Solutions to these problems have been sought, first, by offloading the database system from the mainframe computer to a single, dedicated backend computer. The backend computer has its own disk storage, is used to perform all of the database operations, and interacts with the mainframe. However, database systems with this software single-backend approach still encounter the performance problems when either the backend usage or database applications increase.

The software multiple-backend approach to database management and hardware upgrade is therefore proposed to overcome the performance-gains and capacity-growth problems of either traditional mainframe-based database systems or conventional software single-backend database systems. In this paper we specify the design requirements and issues of the software multi-backend database systems. We show how these requirements and issues affect the design and implementation of a multi-backend database system known as MBDS. Since MBDS is designed specifically for performance gains, capacity growth, and hardware upgrade, we benchmark MBDS in order to verify whether its design and implementation can indeed relate the gains and growth directly to the multiplicity of backends in terms of the response-time reduction and invariance.

1. INTRODUCTION

In this introduction, 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 traditional approach to database management and its hardware upgrade. We then point out another conventional approach to database management and its hardware upgrade. Finally, we motivate the need for an unconventional approach to database management and hardware upgrade.

1.1. The Traditional Approach to Database Management and its Hardware Upgrade

The traditional approach to database management requires that the database-system software runs as an application program in a mainframe computer. Thus, the database system must share the use and control of the resources with all of the other applications of the mainframe computer.

IBM's IMS and SQL/DS, Sperry Univac's DMS-1100, RTI's INGRES and Oracle's Oracle are all examples of database systems using the traditional approach to database management. (See Figure 1.) However, the major drawback of this approach to database management is that as the workload of the mainframe computer increases, the performance of the database system degrades. Whether the increase of the workload is in non-database applications or database applications or both, the performance degradation of a database system is due mainly to the lack of the database system's direct control and exclusive use of the mainframe computer resources. Even if the mainframe computer is upgraded, the additional resources cannot be used directly to lessen proportionally the performance degradation of the database system, since the new resources are shared by other applications and controlled by the mainframe computer.

1.2. The Software Single-Backend Approach to Database Management and its Hardware Upgrade

One conventional approach to the problems of performance degradation and resources sharing and control is to offload the database-system software from the mainframe computer to a separate, dedicated computer with its own disk system. This approach, called the software single-backend approach, was originated by Bell Laboratories in their work on XDMS. As quoted below, the main goals of XDMS were to:

1. obtain a cost saving and a performance gain through specialization of the database operations on a dedicated backend processor;
2. allow the use of shared databases by different mainframe computers, now called hosts;
3. provide centralized [i.e., physical] protection of the databases, and
4. reduce the complexity when developing software for a stand-alone and new machine.

As in the software-laden database systems, the hardware-assisted database systems also utilize the software single-backend approach by off-loading the database-systems.

* The work reported herein is supported by grants from the Department of Defense STARS Program and from the Office of Naval Research and conducted at the Laboratory for Database Systems Research, Naval Postgraduate School, Monterey, CA 93943.
software to a special-purpose computer. Such is the case of the Britton-Lee IDM-500. (See Figure 2.)

In general, software single-backend systems can achieve goals 2, 3, and 4, but have had difficulty in meeting goal 1 entirely. Single backends may be cost-effective, since the cost of a backend may be more than compensated by the cost differential between the new mainframe computer and the existing mainframe computer. In other words, the cost of the new mainframe computer minus the resale value of the existing mainframe computer may be greater than the cost of a single, new backend. Nevertheless, these software single-backend systems suffer from performance problems: in fact, they suffer from the same performance problems of the database systems running on the mainframes. As the use of a software single-backend database system increases, the single backend can no longer maintain the desired performance which had been gained by offloading the database software from the mainframe and by utilizing the dedicated and specialized hardware. Like the hardware upgrade of mainframe computers, the standard approach to the hardware upgrade of a software single-backend system is to use the next more powerful single backend. Unfortunately, such an upgrade does not yield precise, direct and proportional performance gains with respect to cost differentials. Therefore, we need an "unconventional" approach to database management and its hardware upgrade.

1.3. The Software Multiple-Backend Approach to Database Management and its Hardware Upgrade

To overcome the performance problems and upgrade issues of the traditional mainframe-based approach and of the conventional software single-backend approach, the use of multiple backends for the database operations, as an unconventional approach, is being considered. This approach, known as the software multiple-backend approach, may overcome both the performance failings and upgrade issues of the traditional mainframe-based and the conventional software single-backend approaches.

The major emphasis of the software multiple-backend approach is to provide performance gains through specialization of the database operations on dedicated, multiple backends. Unlike XDMS, a software multi-backend database system does not restrict itself to a single backend, but, instead, utilizes multiple backends connected in a parallel fashion in order to achieve performance gains and capacity growth. These backends have identical and replicated software and their own disk systems. In a software multiple-backend configuration, there is a backend controller (i.e., master) which is responsible for supervising the execution of database transactions and for interfacing with the hosts and users. The backends (slaves) perform the database operations with the database stored on the disk systems of the backends. The controller and backends are connected by a communications bus. Users access the system either by way of the hosts or through the controller directly. (See Figure 3.)

The two goals of a software multi-backend database system are of course to overcome the performance problems and upgrade issues of the traditional mainframe-based or the conventional software single-backend database systems. First, by increasing the number of backends, while the size of the database and the size of the responses to the transactions remain constant, the database system is to produce a reciprocal decrease in the response times of the user transactions. Second, by increasing the number of backends proportionally to the increase of transaction responses, the database system is to produce invariable response times for the user transactions. The first goal software the multiplicity of the backends of the database system to be directly related to the performance gains of the database system in terms of the response-time reduction. The second goal enables the multiplicity of the backends of the system to be directly related to the capacity growth of the system in terms of response-time invariance.

In this paper, we present the design, implementation and performance of a multi-backend database system, known as MBDS, which is based on the software multiple-backend approach to database management and hardware upgrade. There are two major focuses. First, we focus on how MBDS is designed and implemented to meet the goals of the software multiple-backend approach to database management and hardware upgrade. This has been the first primary objective in designing, implementing and experimenting with MBDS. Second, we verify whether MBDS can meet the
goals of performance gains and capacity growth of the software multiple-backend approach to database management and hardware upgrade in terms of response-time reductions and response-time invariances. This has been the second primary objective of this research.

Although as a secondary objective we have focused on how software engineering techniques can be applied to the design and implementation of a database system such as MBDS, we will not report our experience on the application of software engineering techniques herein. For a comprehensive discussion of the software engineering techniques used in the design and implementation of MBDS, the interested reader is referred to He\textsuperscript{16} and Orooji\textsuperscript{19}. It suffices to say that MBDS has been carefully and methodically designed and implemented with our best understanding of modern software engineering techniques.

The remainder of this paper is organized as follows. In Section 2 we examine the design requirements and design issues of a multi-backend database system. In Section 3 we present an overview of the MBDS implementation, focusing on the data model, the data language, the data management functions, the directory data, and the placement algorithm of the database. In Section 4 we describe the performance evaluation of MBDS. Finally, we summarize and conclude this paper in Section 5.

2. THE DESIGN OF A MULTI-BACKEND DATABASE SYSTEM

How can a multi-backend database system be designed and implemented to meet the requirements of the software multiple-backend approach to database management and hardware upgrade? In this section we present an analysis of the strategies and decisions used in the design and implementation of MBDS. In the following subsections the necessary and sufficient features of a "good" multi-backend database system are given. These are the design requirements, which are the ultimate goals of the software multiple-backend approach. The characteristics that the multi-backend database system must have in order to satisfy the major design requirements are also given. These are termed design issues.

2.1. Design Requirements

We identify three design requirements that underscore the multi-backend database system. The first requirement states that multi-backend database system must be expandable, in order to support the addition of backends for performance enhancement and capacity growth. This expansion must require no modification to the existing database software, no new programming necessary for the expansion, no modifications to the hardware and no major disruption of system activity when additional backends are being incorporated into the system.

The second requirement mandates that both the hardware and software are generic. The hardware of the backends should be typical and readily available (i. e., off-the-shelf) and can be added to the system with minimal interruption of the system activity. This creates a system that permits a smooth and ready expansion without relying on costly, atypical, special-purpose hardware and without noticeable system interruption. The backend software should be designed so that a new backend can be integrated into the system by simply replicating the database system software of another backend into the new backend. With this requirement, a multi-backend database system can be upgraded by adding new backends of the same type and by using existing system software.

The third requirement suggests that, for storage, a database is evenly distributed across the disk systems of the backends, and, for operation, there are parallel and concurrent processing of transactions by the backends. Thus, when a transaction is being processed, a backend works on its own portion of the database in parallel with other backends working on their own portions of the same database. This is parallel processing of a transaction and parallel access to the database. In addition to parallel processing and access, the backends must process several transactions con-
currently in order to overcome any idling of the backends and any delay in accessing the database. By exploiting the parallelism and concurrency of the backends and by distributing a database evenly for storage, the system should gain in performance. Intuitively, an evenly distributed database may lead to an even distribution of access and processing among the backends. If the number of backends in the system is large, then database access and transaction processing on each backend will be small. Performance gains (in terms of response-time reduction) and capacity gain (in terms of response-time invariance) of a multi-backend database system are likely to be in proportion to the number of backend of the system.

2.2. Design Issues

There are several issues which must be resolved in order to meet the design requirements of a multi-backend database system. In particular, these design issues involve the specification of the characteristics of the backend controller, the communications bus, the backends, the communications traffic, the database store, the data model, the data language, the directory strategy, and the directory placement.

The first issue concerns the backend controller. The overall design goal of a backend controller should focus on minimizing the work done by the controller. (See Figure 3 again.) The controller receives a user transaction either from a host or through a terminal and sends the transaction to all of the backends for execution. The controller also collects all of the results produced by the backends for the user transaction and routes the results to the host or to the terminal. As such, the controller becomes a prime candidate for the bottleneck of the system. By minimizing the work of the controller, and by offloading all of the database management operations to the backends, the controller may reduce the possibility of becoming the system bottleneck. Overall, the functions of the controller are reduced to the pre-processing of the user transactions, the post-processing of the transaction results, the sending and receiving of data from the backends and the hosts, and the arbitration of data insertion into the database.

The second design issue addresses the characteristics and functionality of the communications bus between the controller and the backends. Consider two extreme choices: one where a broadcast bus is shared by the controller and backends or another where a point-to-point high-speed bus between the controller and each backend is utilized. While the high-speed bus may offer a higher communications rate, the broadcast bus is a cost-effective solution, since data-intensive transfers are between the backends and their disk systems and are not between the backends and the controller (i.e., not on the broadcast bus). The choice of the broadcast bus as a cost-effective and efficient solution for both backend communication and backend addition may be warranted.

The third class of issues involves the backends of the system. We require that the backends of the system all have identical software to allow replication of the software on a new backend. Additionally, the backends must have complete software to perform all of the database management functions. These functions include directory management, concurrency control, record processing, and communications. The directory management function is responsible for managing indices, calculating record clusters, allocating the secondary storage addresses for record insertion, maintaining the secondary-storage tables of indices, cluster numbers, and addresses, processing transactions against the directory tables, and providing record addresses for subsequent database access operations. The concurrency control function oversees various accesses to the directory tables and the user data and facilitates the concurrent execution of transactions. The record processing function is used to stage the user data from the secondary storage to the primary memory, to process the staged data, to store data onto the secondary storage, and to return the responses to the controller. Finally, there are communication functions in each backend to control communications among backends and between the backend and the controller. It is necessary to minimize the communications among backends, in order to reduce the communications traffic among them.

The fourth design issue concerns the database. In a multi-backend database system, a database must be placed on the secondary storage in such a way so that all of the subsequent accesses to the database will result in block-parallel and record-serial operations. In other words, all of the backends are accessing, in parallel, the secondary-storage blocks of the same database in their respective disk systems, although the records in the blocks which may satisfy the same transaction or different transactions are being accessed by the backends serially. Thus, the issue really focuses on how to ensure an even distribution of the user database across the disk systems of the backends. Such a distribution requires a data placement algorithm. To achieve an even distribution of data, there must be a processor in the multi-backend database system that is responsible for overseeing the record-insertion process. The controller has an overview of the entire system, and is the logical choice for arbitrating the record insertion process, i.e., controlling the data placement.

The fifth design issue is on the choice of a data model and data language. The chosen data model should easily support the required data distribution and the data placement of the database. The data language for the system is of course based on the chosen data model. It must capture all of its primary operations of the database system.

The sixth design issue focuses on minimizing the communications traffic of the system. The controller should only communicate with the backends for sending the pre-processed user transaction, for arbitrating the data placement, and for receiving results. The backends should execute user transactions autonomously, and only communicate with the controller when results for the user transactions are to be sent. Communication among backends should be held to a minimum, preferably, none.

The seventh and final design issue deals with the directory placement strategies. Should we store the directory on the disks of the controller? Definitely not, since this requires additional communication between the controller and backends and additional work on the part of the controller, impinging on the first and sixth design issues of the system. Should one of the backends be used to maintain the entire directory? Again, such use contradicts the third and sixth design issues of the system. The choice of duplicated directory data at each backend may be considered. By having each backend maintain its own copy of the directory data, much of the extra communications traffic associated with directory processing may be minimized. This strategy also
promotes autonomous, parallel access of directory data by backends, resulting in database operations that are in either single-transaction and multiple-data-accesses or multiple-transactions and multiple-data- accesses modes.

3. THE SYSTEM ARCHITECTURE AND ELEMENTS

In this section we provide an overview of system architecture and elements of MBDS. We also attempt to correlate our design and implementation decisions with respect to the design requirements and issues covered in Section 2. We preface the following subsections with an overview of the MBDS architecture.

When a transaction is received from a host computer or from a terminal, the controller broadcasts the transaction to all the backends. Each backend has a number of dedicated disk drives. Since the data is distributed across the backends, a transaction can be executed by all backends simultaneously. Each backend maintains a queue of transactions and schedules a transaction for execution independent of the other backends, in order to maximize its access operations and to minimize its idle time. Thus, transactions are also executed in the backends concurrently. As required by the software multiple-backend approach, the controller does very little work. It is responsible for broadcasting, routing, and assisting in the insertion of new data. The backends do all of the primary database operations.

Communication between computers in MBDS is achieved by using a time-division-multiplexed bus called the parallel communication link (PCL). MBDS provides a software abstraction to this bus for each computer in order to emulate broadcast capabilities. The abstraction consists of two complimentary processes. The first process, get-pcl, gets messages from other computers off the PCL. The second process, put-pcl, puts messages on the bus to be broadcasted to other computers. Every computer, whether it is the controller or part of a backend, has its own get-pcl and put-pcl processes.

MBDS meets the three design requirements outlined in Section 2.1. MBDS is expandable and capable of auto-configuring to a specific number of backends in the system. Additionally, the backend hardware and software are generic and identical, making it possible to add new backends into the system by simply duplicating the system hardware and replicating the system software. Finally, even distribution of the database is also achieved, as will be elaborated on in Section 3.2.2.

In the rest of this section, we concentrate on major elements of MBDS and begin by describing the data structures and the primary operations of MBDS. We then provide a brief examination of the placement of data in MBDS. Finally, we describe the MBDS process structure.

3.1. The Data Structures and the Primary Operations

Since the focus here is on the design, implementation and performance of MBDS, we will not provide the formal specification of the data structures and primary operations of MBDS. Instead, we will present an informal characterization of them.

3.1.1. The Database Structure

The database structure of MBDS is expanded on the attribute-based data model proposed by Hsiao, extended by Wong, and studied by Rothen. 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 consists of a collection of files. Each file contains a group of records that are characterized by a unique set of directory keywords. A record is composed of two parts. The first part is a collection of attribute-value pairs or keywords. 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, <POPULATION, 25000> is an attribute-value pair having 25000 as the value for the population attribute. A record contains at most one attribute-value pair for each attribute defined in the database. Certain attribute-value pairs of a record (or a file) are called the directory keywords of the record (file), because either the attribute-value pairs or their attribute-value ranges are kept in a directory for identifying the records (files). Those attribute-value pairs which are not kept in the directory are called non-directory keywords. The rest of the record is textual information, which is referred to as the record body. An example of a record is shown below:

```
( <FILE, USCensus>, <CITY, Monterey>,
  <POPULATION, 25000>, {Temperate climate})
```

The angle brackets, <>, enclose an attribute-value pair, i.e., keyword. The curly brackets, {}, include the record body. The first attribute-value pair of all records of a file, by convention, is the same. In particular, the attribute is FILE and the value is the file name. A record is enclosed in the parenthesis. For example, the above sample record is from the USCensus file.

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, e.g., POPULATION :> 20000 is a keyword predicate. More specifically, it is a greater-than-or-equal-to predicate. Combining keyword predicates in disjunctive normal form characterizes a query of the database. The query

```
(FILE = USCensus and CITY = Monterey) or
(FILE = USCensus and CITY = San Jose)
```

will be satisfied by all records of the USCensus file with the CITY of either Monterey or San Jose. For clarity, we also employ parentheses for bracketing conjunctions in a query.

3.1.2. 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

546
mutually exclusive. Now the notion of a cluster can 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), examples of which are given in Figure 4. The attribute table maps directory attributes to the descriptors defined on them. A sample AT is depicted in Figure 4a. The descriptor-to-descriptor-id table maps each descriptor to a unique descriptor id. A sample DDIT is given in Figure 4b. The cluster-definition table maps descriptor-ids 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 4c. Thus, to access the user data, MBDS must first access directory data via the AT, DDIT, and CDT.

There are three classifications of descriptors. A type-A descriptor is a conjunction of a less-than-or-equal-to predicate and a greater-than-or-equal-to predicate, such that the same attribute appears in both predicates. For example, ((POPULATION < 10000) and (POPULATION > 15000)) is a type-A descriptor. A type-B descriptor consists of only an equality predicate. (FILE = USCensus) is an example of a type-B descriptor. Finally, a type-C descriptor consists of the name of an attribute. The type-C attribute defines a set of type-C sub-descriptors. Type-C sub-descriptors are equality predicates defined over all unique attribute values which exist in the database. For example, the type-C attribute CITY forms the type-C sub-descriptors (CITY = Cumberland), (CITY = Monterey), (CITY = Toronto) and (CITY = Columbus), where "Cumberland", "Monterey", "Toronto" and "Columbus" are the only unique database values for the CITY.

At this point, we would like to make some observations on the attribute-based data model. First, we note that the attribute-based data model is data independent. None of the constructs presented above (i.e., database, file, record, attribute-value pair, etc.) are dependent on a specific implementation. Second, despite the strong data independence of the attribute-based data model, the model allows the implementor to take advantage of certain constructs for system optimization. For instance, a natural implementation of the attribute-based data structure could involve indexing on directory attributes, keyword predicates, and groups of keyword predicates. Finally, this model is simple, consisting of a small number of notions and constructs. The attribute-based data model therefore meets the fifth and seventh design issues specified in Section 2.2.

3.1.3. The Primary Database Operations

The attribute-based data language (ABDL), defined by Banerjee [1], is used as the basis of the data language of MBDS. The ABDL supports the five primary database operations, INSERT, DELETE, UPDATE, RETRIEVE, and RETRIEVE-COMMON. A request in the ABDL is a primary operation with a qualification. A qualification is used to specify the part of the database that is to be operated on. Two or more requests may be grouped together to form a transaction. Now, let us illustrate the five types of requests and forget their formal specifications.

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 being inserted. Example 3.1 contains an INSERT request that will insert a record into the USCensus file for the city Cumberland with a population of 40,000.

Example 3.1: INSERT (< FILE, USCensus >, < CITY, Cumberland >, < POPULATION, 40,000 >)

A DELETE request is used to remove record(s) from the database. The qualification of a DELETE request is a query. Example 3.2 is a request that will delete all records whose population is greater than 100,000 in the USCensus file.

Example 3.2: DELETE ((FILE = USCensus) and

POLULATION > 100,000)

An UPDATE request is used to modify records of the database. The qualification of an UPDATE request consists of two parts, the query and the modifier. The query specifies which records of the database are to be modified. The
modifier specifies how the records being modified are to be updated. Example 3.3 is an UPDATE request that will modify all records of the US Census file by increasing all populations by 5,000.

Example 3.3. UPDATE (FILE = US Census) (POPULATION = POPULATION + 5,000)

In this example, (FILE = US Census) is the query and (POPULATION = POPULATION + 5,000) is the modifier.

The RETRIEVE request is used to retrieve records of the database. The qualification of a retrieve request consists of a query, a target-list, and a by-clause. The query specifies which records are to be retrieved. The target-list consists of a list of output attributes. It may also consist of an aggregate operation, i.e., AVG, COUNT, SUM, MIN, MAX, on one or more output attributes. The optional by-clause may be used to group records when an aggregate operation is specified. The RETRIEVE request in Example 3.4 will retrieve the city names of all records in the US Census file with populations greater than or equal to 50,000.

Example 3.4. RETRIEVE ([[FILE = US Census] and (POPULATION ≥ 50,000)]) (CITY)

(FILE = US Census) and (POPULATION ≥ 50,000) is the query and CITY is the target-list. There is no use of the by-clause or aggregation in this example.

Lastly, the RETRIEVE-COMMON request is used to merge two files by common attribute-values. Logically, the RETRIEVE-COMMON request can be considered as two retrieve requests that are processed serially in the following general 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 retrieve request). Example 3.5 is a RETRIEVE-COMMON request which will find all records in the Canada Census file with population greater than 100,000, find all records in the US Census file with population greater than 100,000, identify records of respective files whose population figures are common, and return the two city names whose cities have the same population figures.

Example 3.5. RETRIEVE ([[FILE = Canada Census] and (POPULATION > 100,000)]) (CITY) COMMON (POPULATION, POPULATION) RETRIEVE ([[FILE = US Census] and (POPULATION > 100,000)]) (CITY)

ABDL provides five seemingly simple database operations, which are nevertheless capable of supporting complex and comprehensive transactions. We believe that ABDL can meet the requirements of a data language of primary operations as postulated in Section 2.2.

3.2. The Placement of Data

In this section we consider how data is placed in the multi-backend database system. There are two types of data that must be placed: directory data and user data. For directory data, we adopt the strategy of replicating the directory data at each backend. For user data, we do not replicate any data, instead, we distribute the data evenly across all of the backends.

3.2.1. The Directory Placement

With respect to the multi-backend database system architecture, each backend maintains its own copy of the directory tables. We have adopted the strategy of replicating the directory information as proposed in the seventh design issue. (See Section 2.2 again.) The directory tables are identical up to, but not including, the record ids. In other words, record ids in the CDT are different from one backend to another backend and unique to a backend because they are the secondary-storage addresses of the records stored in the backend's disks. The directory tables are stored in the secondary storage and are staged into the primary memory when needed. There are three possible problems that are identified with this approach.

First, there may be a need to broadcast certain elements of the directory among backends to speed up the execution of a transaction. In particular, when there is a large number of predicates in a query, it may be advantageous to use the replicated directory by having each backend to work on a mutually exclusive set of predicates, and then to exchange results with other backends. While this strategy reduces the time for the directory search, it increases the communications traffic among backends. (See the sixth design issue in Section 2.2.) The optimal solution is of dual approaches. With a large number of predicates, each backend is given a set of distinct predicates. The backends do the directory search for the distinct predicates on the replicated directory in parallel. If a small number of predicates is in a query, the entire query is broadcasted to every backend. The backends do the directory search for the entire query in parallel.

The usual size of a directory for a database is in the range of ten to twenty percent of the entire user data. The replication of the directory data attempts to achieve a reduced access time of the user data by limiting the search space of the database. The only possible drawback with this approach is the increased necessary storage for the replicated directory at each backend. In contrast to the sheer size of and frequent access to the user data, this becomes a minor concern given the reduction of the database access time and the relatively small size associated with the directory.

Our final concern with this strategy is the possibility that the directory data may be dynamically updated in the course of transaction execution. Our solution produces the specification of directory concurrency control mechanisms. This solution is discussed in Section 3.3.3.

3.2.2. The Database Placement

The cluster-based database placement is arbitrated by the controller and carried out by the backends. New clusters are formed by the backends. When a new record is to be included in its cluster, the controller decides which backend will insert the new record into the cluster. The record insertion into the cluster is accomplished by the chosen backend with the placement of the new record on a block of the backend's secondary storage. Under the direction of the controller, the chosen backend will continue to place additional new records of the same cluster in the block until the block of the secondary storage is filled. When this occurs, the backend notifies the controller that the block is full. The
controller then directs another backend to continue the placement of new records of the same cluster. The controller maintains the identification of the backends whose secondary-storage blocks may be used for the insertion of new records into the existing clusters. In a multiple-backend configuration, the cluster-based database placement algorithm achieves a cluster-parallel-and-record-serial operation for any subsequent access to the database

Let's trace through an example. Suppose that our system has four backends, the average size of a record is 200 bytes, and the size of a block of secondary storage is 4K (so each block contains approximately 20 records). A new cluster of 100 records, say C, is defined. The controller picks, say, Backend 3, for inserting records of cluster C. Backend 3 will insert 20 records into a block for the cluster C under the direction of the controller. Then the controller will have Backend 4 insert records of cluster C. After Backend 4 has inserted 20 records, the controller will cycle to Backend 1 and continue the round-robin process until all 100 records are placed on the secondary-storage blocks. If there are new records to be included in cluster C at a later time, the controller will then pick Backend 4, since Backend 3 is the last backend used by the cluster in the algorithm. The database placement algorithm of MBDS facilitates subsequent cluster-parallel-and-record-serial operations, which are, of course, equivalent to the block-parallel-and-record-serial operations, as required in the fourth design issue.

3.3. The Process Structure

In addition to the communications processes (i.e., get-pel and put-pel), there are other processes in MBDS. They are described in this section. First we present the test-interface process which allows the user to interact with MBDS directly. Next, we review the processes of the controller. Finally, we describe the processes of each backend. In Figure 5 we present an overview of the MBDS process structure.

3.3.1. The Test-Interface Process

The test interface to MBDS is menu-driven. There are three levels of menus. Level 1 corresponds to the system level, i.e., the invocation level for the test interface. Level 2 specifies the three main actions of the test interface: loading a database, generating a database, and executing the request interface. Level 3, entered via the request interface, allows the user to choose a new database to work with, create a new list of test transactions, modify an existing list of test transactions, select transactions from an existing list for execution, select an existing list so that all test transactions on the list may be executed, or specify the display of the results. The test interface consists of approximately 2000 lines of C code with an executable image of 68 kbytes.

3.3.2. 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. (See Figure 5 again.) 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 only occurs at one of the backends. Thus the controller must determine the backend at which the insert will occur, along with certain directory information. (See Section 3.2.2.) Post processing is used to collect all the results of a request (transaction) and forward the results to the user.

In the specification of the controller processes, we satisfy the design goals of a backend controller (see the first design issue in Section 2.2). We minimize the work of the controller and offload all of the primary database operations to the backends of the system. We also minimize the inter-computer communications traffic (from the controller to the backends) by restricting the communications to sending the parsed transactions and arbitrating the record insertion. The controller processes (including communication processes) consist of approximately 3000 lines of C code and occupy an executable image of 85.3 kbytes.

3.3.3. The Processes of Each Backend

In addition to the two communications processes, each backend also consists of three other processes. They are of course different from the controller processes. They are directory management, concurrency control, 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, DDT, and CDT. By traversing the directory tables for a request, directory management is able to determine the disk addresses 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.

Figure 5. The MBDS Process Structure
Concurrently 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. There are three concurrency control mechanisms. Two control access to directory data, more specifically, access to descriptors and clusters. The third mechanism controls access to user data. In particular, the secondary-storage addresses of records. The concepts used to implement the concurrency control mechanisms for MBDS are quite unique. Because of the duplicated directory tables at each backend, we are able to devise concurrency control mechanisms that can operate on each backend autonomously. In addition, the structure of these concurrency control mechanisms also guarantees the serializability of user requests (transactions) as they access the user and directory data. The user-data concurrency control mechanism is explained in Boyne and the directory-data concurrency control mechanisms are covered in Demurjian.

Record processing performs the disk I/O operations and other operations specified by the request. Record processing receives the secondary-storage addresses from directory management, which process the request. The results are then forwarded to the controller. When a RETRIEVE-COMMON operation is specified, the results of the first retrieve operation are buffered at the backend, so that the second retrieve operation can be performed by every backend before the final results are returned to the user via the controller.

There are a number of observations that can be made concerning the process structure of the backend. We note that the functionality of the backends satisfies the third design issue. The software of a backend is complete, and is capable of performing all of the primary database operations. The concurrency control mechanisms are used to facilitate the management and access of the duplicated directory and the non-duplicated database. The communications traffic between the backend and the controller and among backends is also minimized. In particular, the backend only communicates with the controller to route transaction results or to participate in the record insertion process. The backends only communicate with each other during a particular phase of the directory search. Each backend of the system consists of approximately 12,000 lines of C code with an executable image of 250 bytes.

4. THE PERFORMANCE EVALUATION STRATEGY AND RESULTS

In this section, we present the performance of MBDS. First, we analyze the basic benchmark strategy. We then present the preliminary results for the performance evaluation. The benchmark strategy focuses on collecting the response time of requests (transactions) that are processed by the system. To adequately conduct the benchmarking of the system, software is developed to collect timing data. The timing software is bracketed in conditional compilation statements of the system to facilitate an easy transition between a running system with timing software and a running system without timing software. Deliberately, we stop our performance evaluation of MBDS at this preliminary stage. The reasons for this decision will be elaborated upon in the concluding section of this paper.

4.1. The Benchmark Strategy

In this section, we give a description of the test database organization and system configurations used in benchmarking. Next, we examine the request set used to collect the timings. Finally, we review the benchmark tests that are to be conducted. The measurement statistics that are collected, and the evaluation results that are obtained.

4.1.1. The Test-Database Organization and Test-System Configurations

The test database was constructed using a record size of 200 bytes. A total of 24 clusters are defined for the test database. The database is restricted to a maximum of 1000 records per backend. MBDS is limited to three different system configurations for benchmarking. We will elaborate about the restrictions and limitations in the concluding section. Table 1 displays the configurations.

<table>
<thead>
<tr>
<th>TEST</th>
<th>No. of Backends</th>
<th>Records/Backend</th>
<th>Database Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1000</td>
<td>200K bytes</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>500</td>
<td>200K bytes</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>1000</td>
<td>400K bytes</td>
</tr>
</tbody>
</table>

Table 1. The Measurement Configurations

Test A configures MBDS with one backend and one thousand records in the test database. Test B configures MBDS with two backends and one thousand records split evenly between the backends. The transition from Test A to Test B is used to verify the first performance goal. (See Section 1 again.) Tests A and B have 23 clusters each of which contains 40 records and one cluster which contains 80 records. In Test A, all of the records are stored on the single backend. In Test B, each backend stores 20 records for each of the first 23 clusters and 40 records for the last cluster.

Test C also configures MBDS with two backends, but, the size of the database is doubled to two thousand records. The transition from Test A to Test C is used to verify the second performance goal. (See Section 1 again.) Test C has 23 clusters each of which contains 80 records and one cluster which contains 160 records. In Test C, each backend stores 40 records for each of the first 23 clusters and 80 records for the last cluster. Although the database is doubled in Test C, the number of records per cluster at a backend remains the same for both Test A and Test C.

4.1.2. The Test-Transaction Mix

In this section we review the retrieve requests that are used to benchmark MBDS. The queries of the retrievals, shown in Table 2, are a mix of single and double predicates.

<table>
<thead>
<tr>
<th>Request Number</th>
<th>Query of Retrieval Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(INTE1 = 10) or (INTE1 = 250)</td>
</tr>
<tr>
<td>2</td>
<td>(INTE2 = &lt; 250)</td>
</tr>
<tr>
<td>3</td>
<td>(INTE2 = &lt; 500)</td>
</tr>
<tr>
<td>4</td>
<td>(INTE1 = &lt; 1000)</td>
</tr>
<tr>
<td>5</td>
<td>(INTE1 = &lt; 200) or (INTE1 &gt; 801)</td>
</tr>
<tr>
<td>6</td>
<td>(INTE1 = &lt; 400) or (INTE1 &gt; 601)</td>
</tr>
<tr>
<td>7</td>
<td>(INTE1 &lt; 201)</td>
</tr>
<tr>
<td>8</td>
<td>(INTE1 &lt; 401)</td>
</tr>
<tr>
<td>9</td>
<td>(INTE1 &lt; 201) or (INTE1 &gt; 800)</td>
</tr>
</tbody>
</table>

Table 2. The Retrieval Requests
There are two directory attributes and thirty-one non-directory attributes in each record. The directory attributes, INTE1 and INTE2, are integer-valued, and have been used for the cluster definition and formation. INTE1 is defined using 5 attribute-value ranges, while INTE2 is defined using 24 attribute-value ranges. The non-directory attributes are used as fillers for the 200-byte record. The retrieve requests given in Table 2 contain queries with equality and inequality keyword predicates, to narrow the search space when accessing the database records.

In Table 3 we present an analysis of the benchmarks given in Table 2. We focus on specifying two characteristics for each retrieve request in the mix: the number of clusters examined by the particular retrieve request and the volume of the database that is retrieved. The values in Table 3 apply to the three testing configurations, A, B, and C, with one exception. The numbers in parenthesis in the third column represent the number of records retrieved for Test C.

<table>
<thead>
<tr>
<th>Request Number</th>
<th>Number of Clusters Examined</th>
<th>Volume of Database Retrieved</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>2(4) records</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>25%</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>50%</td>
</tr>
<tr>
<td>4</td>
<td>24</td>
<td>100%</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>40%</td>
</tr>
<tr>
<td>6</td>
<td>19</td>
<td>80%</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>20% + 1(2) record</td>
</tr>
<tr>
<td>8</td>
<td>15</td>
<td>40% + 1(2) record</td>
</tr>
<tr>
<td>9</td>
<td>19</td>
<td>40% + 2(4) records</td>
</tr>
</tbody>
</table>

Table 3. The Number of Clusters Examined and the Percent of the Database Retrieved

Briefly, let us review and analyze the intent of each of the retrieval requests. The first retrieve examines a large portion of the database (40%) but retrieves only 2 of the 400 records examined. In this situation, we are trying to evaluate how well MBDS does with retrieval requests that examine a large amount of data, but retrieve only a small amount of data. Retrieve requests 2, 3, and 4 show how well MBDS does when examining large volumes of data for retrieving relatively large amounts of data. Request 2 accesses 25% of the database of which 20% of the accessed data is relevant to the answer of the request. Request 3 accesses 52% of the database, with 96% of the accessed data being relevant. Request 4 accesses and returns the entire database. In requests 2, 3, and 4, we are trying to determine how well MBDS does when almost all of the data that is staged from the secondary storage to the primary storage participates in the answer to a retrieval request. Requests 5 and 6 are used to degrade how well MBDS does when the portion of the database, staged from the secondary storage to the primary storage, corresponds to the relevant data. In request 5 (6), 40% (80%) of the database is accessed and returned. Requests 7 and 9 are used to determine how well MBDS performs when only half of the data that is staged from the secondary storage to the primary storage participates in the answer. Request 7 (9) accesses 40% (80%) of the database, with 50% of the accessed data being relevant. The remaining retrieve. Request 8, also gauges how well MBDS performs when only a portion of the staged data is relevant data. In Request 8, 60% of the database is accessed, with 67% of the accessed data being relevant.

4.1.3. The Measurement Strategy, Formulas and Statistics

The basic measurement statistics used in the performance evaluation of MBDS is the response time of request(s) that are processed by the database system. The response time of a request is the time between the initial issuance of the request by the user and the final receipt of the entire request set for the request. The response times are collected for the request set (see Table 2) for each of the three configurations (see Table 1). Each request is sent a total of ten times per database configuration. The response time of each request is recorded. We determine that ten repetitions of each request produce an acceptable standard deviation. Upon completion of the ten repetitions for a request, we calculate the mean and the standard deviation of the ten response times. There are two main statistics that we calculate to evaluate the MBDS performance claims, the response-time reduction and the response-time invariance.

The response-time reduction is defined to be the percentage reduction in the response time of a request, when the request is executed in n backends as opposed to in one backend and the number of records of the database remains the same. Equation 1 is used to calculate the response-time reduction for a particular request, where Configuration Y represents n backends and Configuration X represents one backend. The response-time reduction is calculated for the configuration pair (A, B). (See Table 1 again.) The configuration pair (A, B) is evaluated for the retrieve requests 1 through 9. (See Table 2 again.)

\[
\text{Response-Time Reduction} = 100\% \times \left(1 - \frac{\text{Response Time of Configuration Y}}{\text{Response Time of Configuration X}}\right)
\]

Equation 1. The Response-Time-Reduction Formula

The response-time invariance is defined to be the invariance of the response time of a request, when the request is executed in n backends containing nx number of records as opposed to in one backend with x number of records. Equation 2 is used to calculate the the response-time invariance for a particular retrieval request, where configuration X represents one backend with x records and configuration Z represents n backends, each with x records, i.e., a total of nx records. The response-time invariance is calculated for the configuration pair (A, C), for the retrieve requests 1 through 9.

\[
\text{Response-Time Invariance} = 100\% \times \left(1 - \frac{\text{Response Time of Configuration Z}}{\text{Response Time of Configuration X}}\right)
\]

Equation 2. The Response-Time-Invariance Formula
4.2. The Benchmarking Results

In this section, we present the benchmarking results of MBDS. In particular, we verify the results in the hope of verifying the performance-gains and capacity-growth goals of the software multiple-backend approach to database management as designed and implemented in MBDS. One final note, the units of measurement presented in the tables of this section are expressed in seconds.

Table 4 provides the benchmarking results of MBDS. There are three parts to Table 4. Each part contains the mean and the standard deviation of the response times for requests 1 through 9, which are outlined in Section 4.1.1. The three parts of Table 4 represent three different configurations of the MBDS hardware and the database capacity. The first part has configured MBDS with one backend and the database with 1000 records on its disk. The second part has configured MBDS with two backends, with the database of 1000 records, split evenly between the disks of the backends. The third part has configured MBDS with two backends and a database doubled to 2000 records, where the disk of each backend has 1000 records.

Given the data presented in Table 4, we can now attempt to verify the two MBDS performance goals. We begin by calculating the response-time reduction for the nine requests. In Table 5 we present the response-time reduction for the data given in Table 4. Notice that the response-time reduction is lowest for Request 1, representing a retrieval of two records of the database. On the other hand, the response-time reduction of Request 4, which retrieves all of the database records, is highest: approaching the upper bound of fifty percent. In general, we find that the response-time reduction increases as the number of records retrieved increases. This seems to meet the goal that even if the request set is larger, with the constant database size the reduction in response times will be at a high level, between 45 and 50 percent. In other words, when doubling the backends for the same database, the response times of the same transactions are reduced by nearly one half. The multiplicity of the MBDS backends seems to relate to the performance gains of MBDS directly.

Next, we calculate the response-time invariance for each of the nine requests. In Table 6 we present the response-time invariances for the data given in Table 4. Notice that the response-time invariance is worst for Request 1, which represents a retrieval of two records of the database. On the other hand, the response-time invariances for the requests which access larger portions of the database, i.e., Requests 4 and 6 have only a small response-time invariance. In general, we found that the response-time invariance decreases as the number of records retrieved increases, i.e., the response time remains virtually constant. Again we seem to have statistics to meet the performance goal that, as the size of the request set increases for the same request, the response-time invariance will be held at a relatively low level of 0.15 or less. In other words, by doubling the backends for twice the amount of responses, the response times of the same transactions are kept at nearly the original level. The multiplicity of the MBDS backends seems to directly relate to the capacity growth of MBDS also.

5. CONCLUSIONS AND FUTURE WORK

In MBDS we have a realization of the software multiple-backend approach to database management and its hardware upgrade. MBDS is therefore designed and implemented to meet the requirements for performance gains, capacity growth, and hardware upgrade. The preliminary performance results seem to indicate that MBDS is able to relate the multiplicity of the backends directly to the performance gains and capacity growth of the system. More specifically, the response-time reduction for performance gain and response-time invariance during capacity growth can be realized with the expansion of the system through the use of duplicated backend hardware and replicated backend software. We are encouraged by the results and plan to continue to explore the capabilities and characteristics of the system with the present hardware configuration.

The present system is configured with a VAX-11/780 (VMS OS) as the controller and two PDP-11 44s (RSX-11M

<table>
<thead>
<tr>
<th>Request Number</th>
<th>Response-Time Reduction (A,B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>36.07</td>
</tr>
<tr>
<td>2</td>
<td>45.14</td>
</tr>
<tr>
<td>3</td>
<td>46.53</td>
</tr>
<tr>
<td>4</td>
<td>48.94</td>
</tr>
<tr>
<td>5</td>
<td>47.27</td>
</tr>
<tr>
<td>6</td>
<td>48.81</td>
</tr>
<tr>
<td>7</td>
<td>45.33</td>
</tr>
<tr>
<td>8</td>
<td>47.79</td>
</tr>
<tr>
<td>9</td>
<td>47.21</td>
</tr>
</tbody>
</table>

Table 5. The Response-Time Reduction Between Configurations A and B.
OS) and their disk systems as the backends. The disk system of each backend can support one or more disk drives. Each DEC 2802 disk drive has a 67-megabyte formatted capacity, a peak transfer rate of 806 kbytes per second and an average access time of 42.5 ms (30 ms average seek time - 12.5 ms average latency time). Communication between computers is accomplished by time-division-multiplexed buses, known as parallel communications links (PCLs). When the design and implementation of MBDS began in 1980 the 32-bit-microprocessor-based computers and the broadcast-based communications devices, although anticipated, were not available. Thus, we decided to work under the limits of the then technology and to plan the eventual migration of the system to the intended technology.

The use of a set of interim hardware did create some benefits and restrictions. On the positive side, we were able to begin our design and implementation work long before the arrival of 32-bit-microprocessors and broadcast buses. We were also able to use a powerful minicomputer, i.e., the VAX-11/780, as a development system: since the controller of MBDS does little work, it does not require a powerful minicomputer to run its software. On the negative side, the use of PDP-11/45 as our backends created a number of problems. Each backend was then limited to 256 kbytes of physical memory and to 64 kbytes of virtual space per process. Both of these limitations seriously affected both the development and the testing of MBDS. In development, we were forced to design overlays to fit each of the backend processes into the restricted virtual memory space. In testing, we were forced to limit our testing parameters and strategies to accommodate the limited physical memory and virtual space. The use of PCLs as communications buses also affected our development and testing. By transferring data at a slow rate, the PCL as a relatively slow bus (1 mbit per second compared to the current broadcasting bus technology of 10 mbit per second) makes our timing results unnecessarily high. Furthermore, we must simulate the broadcast capability with PCLs.

At present, we are working on down-loading MBDS to an initial configuration of eight microprocessor-based, broadcast-bus-connected, and Winchester-drive-supported workstations, with one of the eight being used as the controller and the other seven as backends. Our current work includes using Sun-2/170 workstations (4.2 BSD Unix OS) as both the backend controller and the backend computers. This workstation has the Motorola MC68010 as the CPU with 16 mbytes of virtual space per process and uses Ethernet as the broadcast bus among workstations. The disk drives on the backends are Fujitsu Eagle Winchester-type drives, with a formatted capacity of 380 mbytes. When the system is upgraded and running in the Unix environment, we will then begin a full series of tests to complete the evaluation of MBDS in terms of its performance gains, capacity growth, and hardware upgrade was due to D. K. Hsiao in his equipment proposal for MBDS to Digital Equipment Corporation (DEC) at the end of 1979. The design and analysis of MBDS which began in 1980 was due to J. Menon as part of his doctoral dissertation[12]. The research and development on MBDS has been supported by the Office of Naval Research (ONR) from the beginning. The three interim computers (namely, one VAX-11/780 and two PDP-11/44s), the communications buses (PCLs), the disk drives (RM20s), the terminals (VT100s), the line printer, the tape drive, and other peripherals for MBDS were funded by DEC and ONR. The Ohio State University (OSU) provided an air-conditioned room for the equipment.

The detailed specification and implementation of MBDS were due mainly to A. Orooji as part of his study of modern software engineering techniques and their applications to database-systems design and implementation. The implementation work began in 1981 and was supervised jointly by D. K. Hsiao and D. S. Kerr until 1982 and by D. S. Kerr and A. Orooji until 1983. The design and implementation efforts have been documented[4,6,10]. A larger number of implementors have been acknowledged.

In 1983, MBDS was moved from OSU to the Naval Postgraduate School (NPS), along with all of the hardware except the VAX-11/780, which remained at OSU. NPS provided the moving expenses and a VAX-11/780 for MBDS. D. S. Kerr spent the entire academic year of 1983 at NPS debugging MBDS with the help of S. Demurjian. The benchmarking strategy of MBDS was due jointly to S. A. Demurjian and P. R. Strawser[14]. The benchmarking of MBDS was carried out by R. C. Tekampe and R. J. Watson as part of their Master's thesis work[15]. Some testing tools were developed by J. Kovalchik[16]. Throughout the benchmarking phase, S. Demurjian provided the assistance on MBDS and the supervision of the measurements[17,18]. The new hardware and the next series of benchmarking work will be supported by the Department of Defense STARS program.

ACKNOWLEDGEMENTS

The research and development of the multi-backend database system MBDS, with over 15,000 lines of C code, involved many individuals and organizations. We would like to acknowledge their support, contribution, and roles. The original idea of the software multiple-backend approach to database management for performance gains, capacity growth, and hardware upgrade was due to D. K. Hsiao in his equipment proposal for MBDS to Digital Equipment Corporation (DEC) at the end of 1979. The design and analysis of MBDS which began in 1980 was due to J. Menon as part of his doctoral dissertation[12]. The research and development on MBDS has been supported by the Office of Naval Research (ONR) from the beginning. The three interim computers (namely, one VAX-11/780 and two PDP-11/44s), the communications buses (PCLs), the disk drives (RM20s), the terminals (VT100s), the line printer, the tape drive, and other peripherals for MBDS were funded by DEC and ONR. The Ohio State University (OSU) provided an air-conditioned room for the equipment.

The detailed specification and implementation of MBDS were due mainly to A. Orooji as part of his study of modern software engineering techniques and their applications to database-systems design and implementation. The implementation work began in 1981 and was supervised jointly by D. K. Hsiao and D. S. Kerr until 1982 and by D. S. Kerr and A. Orooji until 1983. The design and implementation efforts have been documented[4,6,10]. A larger number of implementors have been acknowledged.

In 1983, MBDS was moved from OSU to the Naval Postgraduate School (NPS), along with all of the hardware except the VAX-11/780, which remained at OSU. NPS provided the moving expenses and a VAX-11/780 for MBDS. D. S. Kerr spent the entire academic year of 1983 at NPS debugging MBDS with the help of S. Demurjian. The benchmarking strategy of MBDS was due jointly to S. A. Demurjian and P. R. Strawser[14]. The benchmarking of MBDS was carried out by R. C. Tekampe and R. J. Watson as part of their Master’s thesis work[15]. Some testing tools were developed by J. Kovalchik[16]. Throughout the benchmarking phase, S. Demurjian provided the assistance on MBDS and the supervision of the measurements[17,18]. The new hardware and the next series of benchmarking work will be supported by the Department of Defense STARS program.

REFERENCES


553


18. Strawser, P. R., "A Methodology for Benchmarking Relational Database Machines," Ph. D. Dissertation, the Ohio State University, Columbus, Ohio, March 1984.
