Calhoun
Institutional Archive of the Naval Postgraduate School

Calhoun: The NPS Institutional Archive

Theses and Dissertations	Thesis Collection

1992-03

The testing, analysis, and correction of the update operation of a parallel, multi-backend supercomputer.

Williams, Michael A.

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/24067

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE TESTING, ANALYSIS, AND CORRECTION OF THE UPDATE OPERATION OF A PARALLEL, MULTI-BACKEND DATABASE SUPERCOMPUTER

by

Michael A. Williams

March, 1992

Thesis Advisor: David K. Hsiao

T259294

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b RESTRICTIVE MARKINGS			
2a. SECURITY CLASSIFICATION AUTHORITY		3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.			
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE					
4 PERFORMING ORGANIZATION REPORT NUMBER(S)		5 MONITORING ORGANIZATION REPORT NUMBER(S)			
6a. NAME OF PERFORMING ORGANIZATION Computer Technology Curriculum Naval Postgraduate School	6b. OFFICE SYMBOL (If applicable) 37	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School			
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000		7b ADDRESS (City, State, and ZIP Code) Monterey, CA 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 Number

11 TITLE (Include Security Classification)

THE TESTING, ANALYSIS, AND CORRECTION OF THE UPDATE OPERATION OF A PARALLEL MULTI-BACKEND DATABASE SUPERCOMPUTER

12. PERSONAL AUTHOR(S)

13a TYPE OF REPORT Master's Thesis	13b TIME COVERED From To	14 DATE OF REPORT (year, month, day) March 1992	15 PAGE COUNT 34

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 COSATI CODES			18 SUBJECT TERMS (continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUBGROUP	Multi-Backend Database Supercomputer, Parallel Update Database Operation

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

The Multi-Backend Database Supercomputer (MBDS) is designed to provide high-performance database management parallely for applications with very large and growing databases. This thesis is a testing, analysis of and correction of the primary database operation UPDATE of MBDS. We provide an overview of the entire MBDS system and then focus on the parallel UPDATE operation in an attempt to discover and correct the deficiencies of the original UPDATE algorithm.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS REPORT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL David K. Hsiao	22b. TELEPHONE (Include Area code) (408) 646-2253	22c OFFICE SYMBOL CS/DB

DD FORM 1473, 84 MAR   83 APR edition may be used until exhausted   SECURITY CLASSIFICATION OF THIS PAGE
All other editions are obsolete   Unclassified

THE TESTING, ANALYSIS, AND CORRECTION OF THE UPDATE OPERATION OF A
PARALLEL, MULTI-BACKEND SUPERCOMPUTER

by

Michael A. Williams
Lieutenant, United States Navy
B.S., University of Mississippi, 1985

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1992

ABSTRACT

The Multi-Backend Database Supercomputer (MBDS) is designed to provide high-performance database management parallely for applications with very large and growing databases. This thesis is a testing, analysis of and correction of the primary database operation UPDATE of MBDS. We provide an overview of the entire MBDS system and then focus on the parallel UPDATE operation in an attempt to discover and correct the deficiencies of the original UPDATE algorithm.

TABLE OF CONTENTS

iv

I. AN INTRODUCTION
TO
THE PARALLEL DATABASE COMPUTER ARCHITECTURE

A database management system (DBMS) must provide fast, accurate and efficient information processing. A today's DBMS is only adequate for current information processing require-ments, but not adequate for new applications, such as multi-media data being utilized in the insurance industry where the multimedia database is several orders of magnitude bigger than the largest databases found today. For new applications data-bases larger than a terabyte (10^{12} bytes) will not be unusual. The current DBMS architecture cannot be scaled to such magnitudes and operations on the very large databases. Conducting set-oriented database operations in a parallel DBMS architecture is an area that has shown increasing promise in solving this problem.

A. PARALLEL ARCHITECTURE FOR DATABASE MANAGEMENT

Conducting parallel operations in a supercomputer for increasing the speed of computations is not a new idea. There are numerous production-level, numerical-oriented supercomputers. However, this type of numerical supercomputers is not effective with the storage and retrieval of a very large database.

An experimental supercomputer for database operations, known as the Multibackend Database Supercomputer (MBDS), has been developed to provide parallel database operations. This prototype system is a research vehicle located in the Laboratory for Database Systems Research at NPS and is utilized for the study of the design and performance of the parallel and scalable database supercomputer.

The basic motivation of MBDS is to provide an architecture that spreads the work of DBMS among multiple backends (dedicated computers), each of which executes the same system software in parallel, thus drastically improving the DBMS performance. (Hsiao, 1983,p.302)

MBDS is presently configured with eight parallel database processors (backends), each of which has three disk drives - a smaller one for paging programs, a small one for meta data, and a larger one for the base data of the database. The architecture of MBDS is illustrated in Figure 1.

MBDS provides the necessary conditions for database management performance gains and capacity growth through parallel database management operations. (Hsiao, 1991)

MBDS is considered in two major sections, the controller section and the backend section. A discussion of the software for each follows.

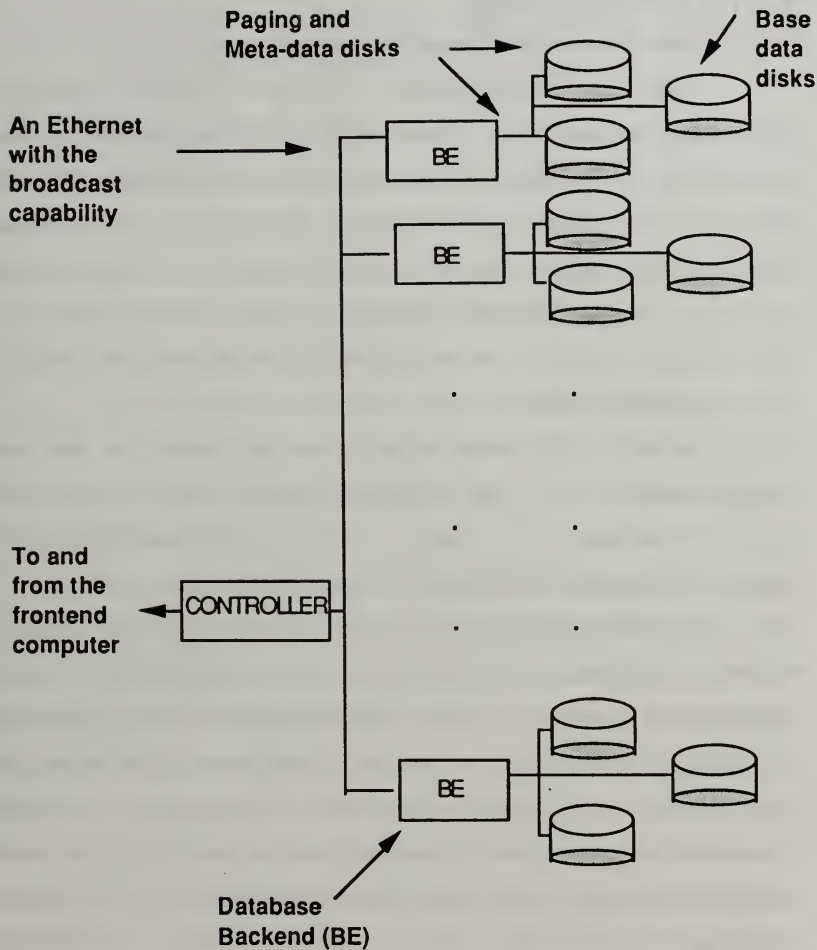**Paging and Meta-data disks**

Base data disks

An Ethernet with the broadcast capability

BE

BE

To and from the frontend computer

CONTROLLER

BE

Database Backend (BE)

FIGURE 1. THE MULTIBACKEND DATABASE S 'PERCOMPUTER (MBDS)

3

B. THE SOFTWARE OF MBDS

1. The Controller-Computer Software

The Controller consists of the following five main processes: Request (or Transaction) Processing (TP), Post Processing (PP), Insert-Information-Generator (IIG), PUT, and GET. TP interfaces with the user software of the system, identifies each user request, pre-processes the request and broadcast the pre-processed request to all the backends. Each backend computer in turn places the broadcasted request in it's request queue.

PP also interfaces with the user software. It performs post-processing on the database-transaction results and provides the results to the user. PP interfaces with TP which allows it to properly identify the intended user.

IIG controls the insertion of records onto a backend's database store and is responsible for the even distribution of each record cluster into the database stores of the backends. IIG maintains the space utilization table which provides the disk track information required to maintain an even distribution of records clustered. The space utilization table keeps the following information up-to-date for each cluster of records:

(1)   Identifies the backend whose database store contains the first trackfull of records of the cluster.

(2)   Identifies the backend whose database store contains the last trackfull of records of the cluster.

4

(3)   Identifies the backend whose database store can
        provide the first available trackfull of storage for
        inserting new records of the cluster.

Finally, PUT and GET provide the communications link among computers, i.e., the controller and backends. PUT places messages on the LAN for transmission to other computers via either the one-to-one or the broadcasting mode. GET receives messages from the other computers via the LAN.

2.   The Backend-Computer Software

In a backend computer, there are five processes that control all the backend operations. They are Directory Management (DM), Record Processing (RP), Concurrency Control (CC), GET, and PUT.

The Directory Management process handles the managing of meta-data. Meta-data is stored information about the base data. Collectively, the three meta-data constructs form the directory of the database. They are attributes, descriptors, and clusters. An attribute is used to represent a category or certain common property of the base data, e.g., POPULATION. A descriptor is used to describe an unique value or a range of values that an attribute can have. For example, (1000 < 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 in terms of

5

their values. A cluster is a group of records such that every record in the cluster satisfies the same set of descriptors.

The condition that the descriptors defined for a given attribute have mutually exclusive attribute values is an important one. Mathematically, the descriptors of the attribute serve to derive equivalence classes which effectively partition the database into mutually exclusive sets of records (clusters). These clusters allow for an even distribution of a database onto the backend stores of MBDS.

The Record Processing (RP) process is responsible for managing the base-data of the database. Specifically, RP conducts record retrieval and selection.

Concurrency Control (CC) is responsible for maintaining meta-data and base-data integrity during execution of a user request or transaction. Since the data requirements of a user request may overlap, it is important that data consistency is maintained while request are being processed.

Each backend has a pair of processes for communications, the GET process for getting transactions or messages from LAN and the PUT process for placing responses or messages on LAN. DM, RP, CC, GET, and PUT are the only five processes of a backend. These five processes are replicated in every backend and are supported by an Unix operating system with TCP/IP protocols. Figure 2 illustrates the relationship of the controller processes and the backend processes.

6

From the frontend computer

To the frontend computer

The Controller Computer

TP

PP

IIG

PUT

GET

to other backends

Ethernet (LAN) with Broadcasting Capability

A Backend Computer

GET

PUT

CC

DM

RP

The Meta-Data Store
(disks of the
backend computer)

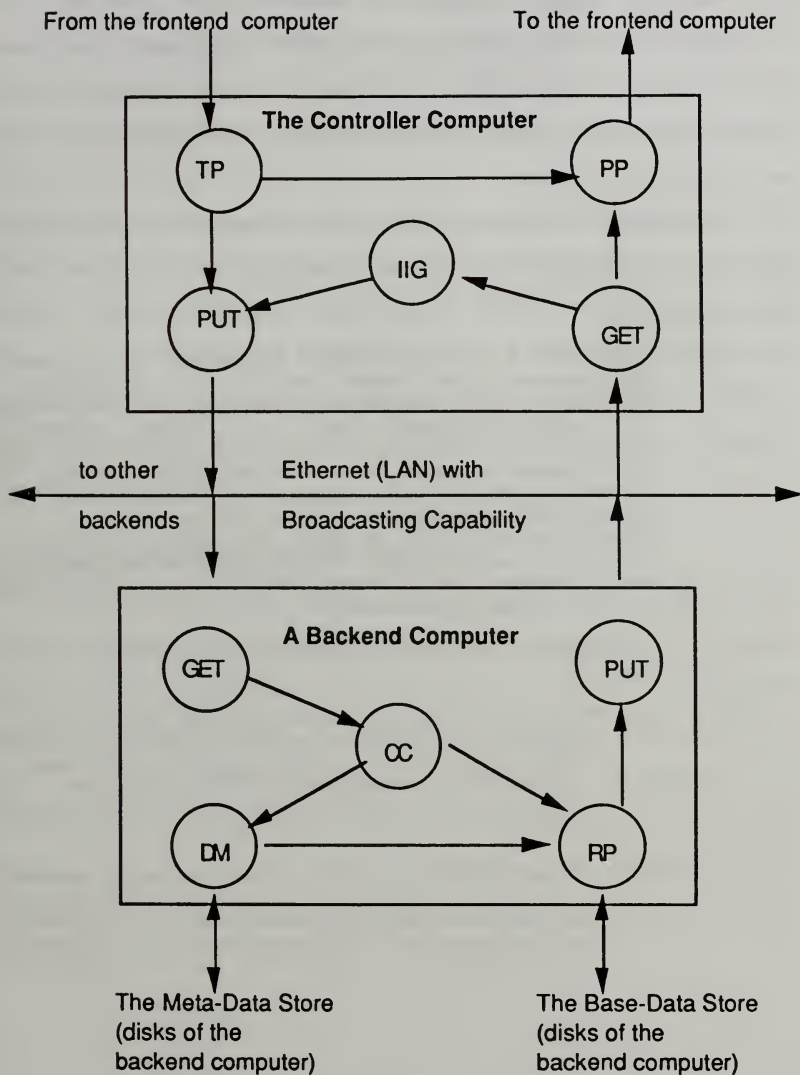The Base-Data Store
(disks of the
backend computer)

Figure 2. The Organization of MBDS Processes

C. THE FIVE PRIMARY OPERATIONS OF MBDS

The five primary operations of MBDS are Insert, Retrieve, Delete, Update, and Retrieve-Common. Insert operates on a single record at a time while the other four operate on a set of records at a time.

A necessary operation for all databases is the Update operation. The Update operation of MBDS is very complex due to the following:

(1)   Update is a multiple-stage operation, i.e., each update must first stage the data into database processors from the database stores for processing, then perform necessary updates of values, delete the original data and return the newly updated data back to database stores. This 4-stage operation must be conducted by all the database processors and their corresponding database stores in parallel. Considerations must be made for coordinations among the parallel processors, buffering require-ments between multi-stage data movements and loads to individual processors.

(2)   Handling of an Update query becomes complicated because:

(a)   Clusters and records which may satisfy the query of an update must be locked with read/write-deny locks because any use of these clusters and records before the update operation is complete may generate erroneous results.

(b)   Clusters and records which are being updated may become new clusters and new records; thus, we must now handle the deletion of old clusters and records and the creation of new clusters and records.

8

D. THE GOAL OF THE THESIS

The current implementation of Update does not work well. It can update small database stores only. The Update operation fails when tested on larger database stores. The cause of this failure is unknown.

The goal of this thesis is to determine the defect(s) in the design and implementation of the Update operation. We must find a theory as to why the original Update does not work and then verify this theory through tests and analyses. Next,there is a need for a theory on how to correct the defective Update operation. Time permitting, we may implement the proposed corrective measures.

E. THE ORGANIZATION OF THE THESIS

The thesis is organized into three chapters in addition to this introduction. In Chapter II, we describe the Update operation logically in the context of the Multibackend Database Supercomputer. In addition, we present theories on why the current Update operation is defective. In Chapter III, we propose new design and implementation in order to overcome defects of the present Update operation. In Chapter IV, we summarize our findings and indicate unfinished work which will require others to carry out.

II.   THE UPDATE OPERATION WITHIN MBDS

Chapter II presents a logical description of the UPDATE operation within the context of the MBDS environment and discusses theories on why UPDATE does not perform well.

A.   BACKGROUND INFORMATION FOR UNDERSTANDING THE UPDATE ALGORITHM

1.   The Base Data Organization

As a database computer, MBDS must have a data model to characterize its database and to allow the user to refer to the database in terms of its logical properties. The data model used for MBDS is the attribute-based data model (ABDM).

Every piece of data in the database is characterized in ABDM as an attribute-value pair. An attribute-value pair is a member of the Cartesian product of the attribute set and the value domain of the attribute. As an example, <POPULATION, 30000> is an attribute-value pair having 30000 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 are called the directory keywords of the record, because either the attribute-value pairs or their ranges are kept in a directory for indentifying records. 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,30000>,

    {Temperate,Climate})

The angle brackets, <,>, enclose an attribute-value pair. The curly brackets, {,}, enclose the record body and the entire record is enclosed within the parenthesis. All the records of the database comprise its base data. Realistically, there are thousands, or even millions, of base data or records, in the database.

    The records of the database are 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 would be POPULATION < 30000. Keyword predicates, combined in disjunctive normal form, comprise a query of the database. The query

    (FILE = USCensus and CITY = Monterey)

will be satisfied by all records of the USCensus file with the City of Monterey.

2. The Meta Data Structure

    To manage the database, MBDS uses meta data which are organized into three tables: attribute table (AT), descriptor-to-descriptor-id table (DDIT), and the cluster-definition

11

table (CDT), examples of which are given in Figure 3. AT maps (directory) attributes of AT to the descriptors defined on them. DDIT maps each descriptor to a unique descriptor id. CDT maps descriptor-id sets to cluster ids. Each entry consists of a unique cluster id, a set of descriptor ids whose descriptors define the cluster, and ids of the records that are in the cluster. Thus, to access the user data, MBDS must first access meta data via the AT, DDIT, and CDT.

   3. **The Distributon of Meta and Base Data On Database Stores**

The distribution of meta data and base data on their separate database stores takes place differently, although both types provide for parallel accessing of data. A description of their differences is provided in the following sections.

   a. The Meta-data Distribution

Meta data are usually one or two orders of magnitude smaller in size than base data. Due to the relatively small size of meta data, the designers of MBDS decided to replicate the meta data onto each backend's database store. Consequently, all the backends can access their own meta-data stores and identical sets of attribute, descriptor, and cluster tables, in parallel (Hsiao,1991).

ATTRIBUTE	ATTRIBUTE TYPE	DDIT ENTRY
POPULATION	A	D11
CITY	C	D21
FILE	B	D31

ATTRIBUTE TABLE (AT)

ID	DESCRIPTOR
D11	0 < POPULATION < 30000
D21	CITY = MONTEREY
D31	FILE = USCENSUS

DESCRIPTOR-TO-DESCRIPTOR-ID TABLE (DDIT)

ID	DESCRIPTOR-ID SET	RECORD-ID
C1	{D11,D21,D32}	R1,R2
C2	{D11,...}	R1,...

CLUSTER-DEFINITION TABLE (CDT)

FIGURE 3

13

b. *The Base-data Distribution*

Base data comprises the bulk of a database. Therefore, they are not replicated for storage. Also, they are stored on each backend's own high-capacity disk drives using the following distribution algorithm:

(1)   From the Cluster table, the controller picks up a cluster identifier and its associated records;

(2)   The controller blocks a variable-size cluster into fixed-size trackfuls of records;

(3)   The controller determines the identifiers of the back ends, each of which can provide a track of available storage for the cluster identified (recall that the controller IIG has a storage utilization map to keep track of such information);

(4)   The controller sends in parallel all the trackfuls of records to the backends identified;

(5)   Each identified backend places its block of one or more trackfuls of clustered records into its base-data store and enters identifiers of records stored onto the replicated CDT entry corresponding to the cluster on the meta-data store.

(6)   The controller then updates its space utilization map with respect to this cluster; and

(7)   The entire procedure is repeated for all subsequent clusters.

This one-track-per-backend database distribution algorithm evenly distributes records of a cluster over a set of separate, parallel database stores. Subsequent accesses to the records of a cluster can now be processed in parallel.

B. THE UPDATE OPERATION ON MBDS

An UPDATE request is used to modify records of the database. The format 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. For example, the following UPDATE request

UPDATE(FILE= USCensus) (POPULATION = POPULATION + 5000) will modify all of the records of the USCensus file by increasing all populations by 5000. In this example, (FILE = USCensus) is the query and (POPULATION = POPULATION + 5000) is the modifier.

1. The Update Algorithm

(a) The Update request is broadcasted by the controller to all the backends.

(b) The Directory Management process on each backend performs descriptor processing and address generation for the Update request. Descriptor processing consists of determining the descriptor ids of the descriptors that satisfy the keywords in the query. This set of descriptors, which satisfies the query, is mapped to the Cluster-definition table (CDT) to determine the appropriate cluster id. Given the cluster id, the record ids are readily available in CDT. The set of selected record ids are passed to the address-generation function which determines the set of track

15

addresses in the secondary storage. These addresses permit accesses to the records required for the Update operation. The set of cluster ids are sent to Concurrency Control to be locked until completion of the Update operation.

(C)   Directory Management passes this set of track addresses to Record Processing for record retrieval and updating. Updating takes place as follows:

(1)   Fetch the entire set of tracks from the database store (secondary memory). Data are staged in the primary memory for quicker accesses during the record-modification phase of the operation.

(2)   Reserve a result buffer (updated records that change clusters are temporarily stored here prior to being sent to the controller for insertion into a new cluster).

(3)   For each address in the set of track addresses, fetch the track from the disk into the track buffer.

(4)   Examine the records in the track buffer one-by-one. If a record is marked for deletion, disregard it. If a record does not satisfy the query of the request, disregard it. If a record satisfies the query of the request, compute the new value according to the modifier and update the record in the track buffer. Send the old and new values of the updated record to DM to determine if the record has changed clusters. If the newly updated record changes cluster, then add the record to the result buffer and mark the old record for deletion in the track buffer.

(5)　After examining all records in the track buffer, store the track buffer back to the disk.

(6)　Flush the result buffer and send the results to the IIG process in the Controller for insertion into a new cluster.

C.　THEORIES ON WHY THE UPDATE OPERATION IS NOT FULLY FUNCTIONAL WITHIN MBDS

1.　Theory One: Erroneous Hashing Techniques

In the Update operation, each backend must first select a set of records to be operated on. Regardless of its size, this record set is retrieved from the database stores and stored in the virtual memory temporarily for subsequent operations. Hashing techniques are used to obtain the addresses for the temporary storage. The hashing technique used may be erroneous, causing nearly all of the records to be stored at the same virtual memory locations.

2.　Theory Two: Erroneous Allocations of Memory

Recall that the current Update algorithm works for small size databases, e.g., 30 record database, but not for larger size databases, say, 500 records or more. Also, consider the fact that MBDS supports a concurrent environment (multiple users active on the system simultaneously). This concurrency capability forces a partitioning of the primary memory throughout the system. Limiting the available memory per user coupled with the fact all modifications for

17

concurrency capability forces a partitioning of the primary memory throughout the system. Limiting the available memory per user coupled with the fact all modifications for an UPDATE is done in the memory leads us to theorize that there may be a possible problem with memory allocation/availabiltiy.

III. AN ALTERNATIVE METHOD
OF
IMPLEMENTING THE UPDATE OPERATION

In this chapter we focus on the theories presented in Chapter II. We provide evidence to substantiate the most promising theory and explain an alternative implementation method to overcome the problem with UPDATE.

A. DISCUSSION OF THEORY ONE: ERRONEOUS HASHING TECHNIQUES

The original design specifications for the UPDATE operation, as discussed in Chapter II, were never fully implemented. In particular, the staging of the entire set of tracks in the virtual memory via a hashing function was never implemented. Instead of retrieving the entire set of tracks to be updated during a single fetch, the UPDATE routine fetches one track at a time and puts it into the virtual memory. Then, performs all the required update steps on the records from this track and stores the newly updated track back to the base data disk.

Therefore, our research efforts in this area led to the validation of theory one; and through test and analysis we discovered and fixed a logic error in the hashing function utilized by the RETRIEVE operation.

The original hashing function utilized by the RETRIEVE operation was hashing all the retrieved records into the same

location in the virtual memory. Our modified version of the hashing function accomplishes the desired goal of a hashing function,i.e., an even distribution of data throughout the memory area.

B.   DISCUSSION OF THEORY TWO: ERRONEOUS ALLOCATIONS OF MEMORY

For theory two we generated several test databases of various sizes, and attempted to run the UPDATE routine on each to determine the point of operational failure.

Initially, the UPDATE operation performed successfully on a database of 30 records. Every UPDATE query used throughout our test and analysis phase was written to update every record in the database. Therefore, a successful run on a database of 30 records, implies 30 records were updated. Next, we tested the UPDATE operation on a database of 35 records and the entire system crashed. Through debugging we discovered the point of system failure. The system crashed while attempting to perform a write routine from the real memory on backend one to the paging disk on the same backend. Trouble-shooting of this problem led to the discovery of the following:

Each backend has only 4 million bytes of the real memory. From this 4 megabytes there are only 1.8 megabytes available for conducting database operations such as UPDATE. Therefore, the code required to perform an UPDATE operation must be paged in and out of the real memory as required. Thus, one use of the paging disk is to partition the executable code of each

backend process into fixed size pages of 1024 bytes and provide a storage area for these pages. A second use of this paging disk is to provide a temporary storage area for a snapshot of the system. In particular, a temporary file is maintained on the paging disk to allow a user to log on, conduct operations, and then log off in the middle of a session without losing his/hers current status of the system.

Since this is a multiple user system, there exists the possibility of having several separate snapshot files residing on the paging disk simultaneously.

The MBDS system crashed while conducting an UPDATE operation on 35 records because the paging disk became full with too many snapshot files left from prior sessions.

A general cleanup of the paging disk, i.e., erasing of all the unnecessary snapshot files can free up enough space to allow a successful UPDATE operation on the 35 record database.

Next we increased the size of the test database to 50 records and performed a successful UPDATE operation. At this point we increased the size of the test database by increments of 50 records and ran the UPDATE operation. Each time the UPDATE operation ran successfully.

Our final test database was 450 records. A successful UPDATE was performed on this database, thus convincing us that the problem was solved. The solution did not lie within the UPDATE algorithm as originally suspected, but within the underlying operating system environment.

IV. CONCLUSIONS

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

A. THE SUMMARY

Performance problems and upgrade issues have always been an obstacle in traditional mainframe-based, single-backend database systems. Never has this been more evident today as the new database application requires a database which is several orders of magnitude bigger than the largest database found in conventional applications. The Multi-Backend Database Supercomputer (MBDS) attempts to overcome this type of problem through specialization and parallelization of the database operations on multiple dedicated backend computers.

A critical tool for any DBMS is the UPDATE operation. The UPDATE operation on MBDS was not fully functional and required further testing and analysis in order to determine its defect.

In this thesis effort, we have first gained a thorough knowledge of the entire MBDS. Secondly, we have acquired a working knowledge of the C programming language in order to understand the MBDS operations. Then, we have developed

22

numerous test databases and conducted operational testing of
the UPDATE routine to determine the point of any malfunction.
Having understood the capabilities of the original UPDATE
operation, we then dissect the UPDATE algorithm and look for
a logic flaw. Once convinced that the algorithm has been
sound, we focus our attention towards the underlying operating
system which supports the UPDATE operation. In particular, we
have looked at two possible problem areas: erroneous hashing
techniques, and erroneous allocations of the memory. A
thorough investigation of both has led us to the following
corrections to MBDS:

- Our modified version of the hashing function is utilized
  to retrieve records into the virtual memory. It now
  supports an even distribution of records throughout the
  memory area.

- The UPDATE operation now performs flawlessly with large
  test databases, no longer crashing the entire system on
  very small databases.

B. DIFFICULTIES ENCOUNTERED

The size of MBDS and the UNIX operating system both
contributed to a 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.
Additionally, the student must become very proficient in the

C programming language in order to understand the MBDS implementation.

Once the student has understood MBDS and the UNIX operating system and becomes proficient in C, the student must develop a theory or theories of possible defects in UPDATE. This has not been a trivial task.

C.   RECOMMENDATIONS FOR FUTURE EFFORTS

The original design specifications for the UPDATE operation allow for the modifier of an UPDATE query to be one of five types:

- Type 0: <attribute = constant>
- Type I: <attribute = f(attribute)>
- Type II: <attribute = f(attribute1)>
- Type III:<attribute = f(attribute1) .of Query>
- Type IV: <attribute = f(attribute1) of Pointer>

Let a record whose attribute is being modified be referred to as the *record being modified*. Then a type-0 modifier sets the new value of the attribute being modified to constant. A type-I modifier sets the new value of the new attribute being modified to be some function of its old value in the record being modified. A type-II modifier sets the new value of the attribute being modified to be some function of some other attribute value in the record being modified. A type-III modifier sets the new value of the attribute being modified to

be some function of some other attribute value in another record uniquely identified by the query in the modifier. Finally, a type-IV modifier sets the new value of the attribute being modifier to be some function of some other attribute value in another record identified by the pointer.

Currently, only the type-0 modifier is implemented within the UPDATE operation. Thus, future work on the UPDATE operation will be to implement the other four types of modifiers for an UPDATE query.

LIST OF REFERENCES

Computer Science Department, Naval Postgraduate School, *A Parallel, Scalable, and Microprocessor-Based Database Computer for Performance Gains and Capacity Growth*, by D. K. Hsiao, 1991.

Hsiao, D. K., *Advanced Database Machine Architecture*, p. 302, Prentice-Hall, Inc., 1983.

Hsiao. D. K., *Modern Database System Architectures*, D. K. Hsiao, 1991.

Hurson, A. R., Miller, L. L., and Pakzad, S. H., *Parallel Architectures for Database Systems*, pp. 31-34, The Institute of Electrical and Electronics Engineers, Inc., 1989.

INITIAL DISTRIBUTION LIST