

AD-A146 534

DESIGN AND ANALYSIS OF AN SQL (STRUCTURED QUERY
LANGUAGE) INTERFACE FOR A MULTI-BACKEND DATABASE SYSTEM
(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA G N MACY

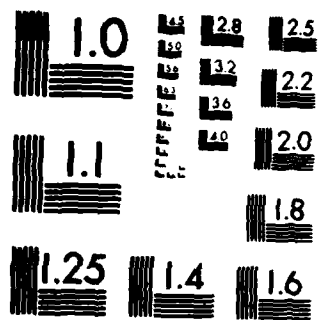
1/1

UNCLASSIFIED MAR 84

F/G 9/2

NL

END
DATE
FILMED
11-84
DTIC



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

2

NAVAL POSTGRADUATE SCHOOL

Monterey, California

AD-A146 534



DTIC
ELECTE
OCT 11 1984
S B

THESIS

DESIGN AND ANALYSIS OF AN SQL
INTERFACE FOR A
MULTI-BACKEND DATABASE SYSTEM

by

Griffin Newton Macy

March 1984

Thesis Advisor:

D. K. Hsiang

Approved for public release; distribution unlimited

DTIC FILE COPY

84 10 10 020

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM	
1. REPORT NUMBER		2. GOVT ACCESSION NO. A146 539	
4. TITLE (and Subtitle) Design and Analysis of an SQL Interface for a Multi-backend Database System		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis March, 1984	
7. AUTHOR(s) Griffin Newton Macy		6. PERFORMING ORG. REPORT NUMBER	
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		8. CONTRACT OR GRANT NUMBER(s)	
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		12. REPORT DATE March, 1984	
		13. NUMBER OF PAGES 89	
		15. SECURITY CLASS. (of this report)	
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited			
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)			
18. SUPPLEMENTARY NOTES			
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Relational Database Systems, SQL, Relational Query Languages			
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) Recent research in the area of database machines has been directed at achieving greater efficiency and increasing user-friendliness. This thesis is concerned with the second of these research directions, increasing user-friendliness. One development toward increased user-friendliness is the growing acceptance of the relational data model and relational query languages. Relational interfaces provide the user with an easy-to-understand (Continued)			

DD FORM 1473

EDITION OF 1 NOV 66 IS OBSOLETE

S/N 0102-LF-014-6601

1

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT (Continued)

data representation and language with which to manipulate the data. This thesis presents the design and analysis of a relational query language interface, using the SQL relational query language, for the Multi-Backend Database System (MDBS), a database machine which uses the attribute-based model. The purpose is two-fold: first, to provide the user with an easier-to-understand language-to-machine interface, thereby making MDBS available to the wider community of relational database users; second, to investigate how the attribute-based model may be used to support relational databases.

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

Approved for public release, distribution unlimited.

Design and Analysis of
an SQL Interface
for a
Multi-backend Database System

by

Griffin Newton Macy
Lieutenant, United States Navy
B.S.M.E., University of Kansas, 1978

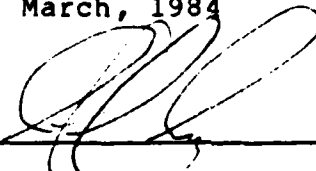
Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

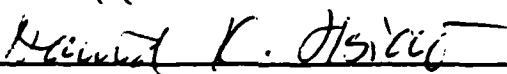
from the

NAVAL POSTGRADUATE SCHOOL
March, 1984

Author:



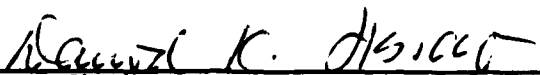
Approved by:



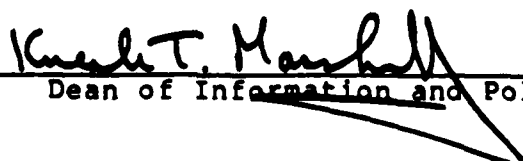
Thesis Advisor



Second Reader



Chairman, Department of Computer Science



Dean of Information and Policy Sciences

ABSTRACT

Recent research in the area of database machines has been directed at achieving greater efficiency and increasing user-friendliness. This thesis is concerned with the second of these research directions, increasing user-friendliness. One development toward increased user-friendliness is the growing acceptance of the relational data model and relational query languages. Relational interfaces provide the user with an easy-to-understand data representation and language with which to manipulate the data.

This thesis presents the design and analysis of a relational query language interface, using the SQL relational query language, for the Multi-Backend Database System (MDBS), a database machine which uses the attribute-based model. The purpose is two-fold: first, to provide the user with an easier-to-understand language-to-machine interface, thereby making MDBS available to the wider community of relational database users; second, to investigate how the attribute-based model may be used to support relational databases.

TABLE OF CONTENTS

I.	INTRODUCTION.....	9
A.	DESIGN GOALS FOR EFFICIENCY IN MDBS.....	10
B.	RELATIONAL QUERY LANGUAGE DEVELOPMENTS.....	13
C.	ORGANIZATION OF THE THESIS.....	16
II.	ORGANIZATION OF THE MULTI-BACKEND SYSTEM.....	17
A.	THE ATTRIBUTE-BASED DATA MODEL.....	19
B.	IMPLEMENTATION OF THE ATTRIBUTE-BASED MODEL IN MDBS.....	23
C.	FUNCTIONS OF THE CONTROLLER.....	24
D.	FUNCTIONS OF THE BACKEND.....	24
III.	THE MDBS QUERY LANGUAGE.....	27
A.	THE RETRIEVE REQUEST.....	27
B.	THE INSERT REQUEST.....	29
C.	THE DELETE REQUEST.....	29
D.	THE UPDATE REQUEST.....	29
IV.	THE RELATIONAL QUERY LANGUAGE, SQL.....	33
A.	THE SELECT REQUEST.....	33
B.	THE INSERT REQUEST.....	38
C.	THE DELETE REQUEST.....	39
D.	THE UPDATE REQUEST.....	40

V.	THE MAPPINGS FROM SQL TO THE MDBS	
	QUERY LANGUAGE.....	41
A.	GRAPHIC NOTATION.....	42
B.	MAPPING REQUESTS FROM SQL TO MDBS.....	43
	1. Mapping the SQL SELECT into the	
	MDBS RETRIEVE.....	45
	2. Mapping the SQL INSERT into the	
	MDBS INSERT.....	46
	3. Mapping the SQL DELETE into the	
	MDBS DELETE.....	48
	4. Mapping the SQL UPDATE into the	
	MDBS UPDATE.....	48
C.	THE CONVERSION MAPPING TO THE MDBS QUERY.....	51
D.	THE CONVERSION MAPPING TO THE MDBS RECORD.....	56
E.	THE CONVERSION MAPPING TO THE MDBS MODIFIER....	58
VI.	RECOMMENDATIONS FOR IMPLEMENTATION.....	65
A.	SQL AND MDBS DIFFERENCES.....	65
	1. The Disjunctive Normal Form.....	66
	2. Differences in the Insert Request.....	66
B.	EXPANDING THE FUNCTIONALITY OF THE	
	INTERFACE.....	67
	1. Implicit Joins Through Nested Selects.....	68
	2. Formatting Options.....	70
	3. Arithmetic Operations and Functions.....	70

C.	JOIN AND SORT OPERATIONS.....	71
D.	TOOLS FOR ACTUAL IMPLEMENTATION.....	72
VII.	CONCLUSION.....	74
A.	THE DIRECT MAPPINGS.....	75
B.	ENHANCEMENTS TO SUPPORT FURTHER MAPPINGS.....	78
C.	OPERATIONS FOR WHICH NO MAPPING EXISTS.....	78
APPENDIX A:	FORMAL SPECIFICATION OF DML FOR ATTRIBUTE-BASED LANGUAGE.....	80
APPENDIX B:	FORMAL SPECIFICATION OF DML FOR SQL MAPPING.....	83
	LIST OF REFERENCES.....	86
	INITIAL DISTRIBUTION LIST.....	89

ACKNOWLEDGEMENTS

I would like to thank the following people.

Dr. David Hsiao for the guidance he provided during the writing of this paper and for the opportunity to learn from him.

A very special thanks goes to Dr. Paula Strawser for her continued help and support in the investigation, writing and editing of this document. Her assistance and direction repeatedly proved to be invaluable and enlightening.

My wife and son, Dixie and Chris, for their everlasting support and faith.

I. INTRODUCTION

The rapid growth in the use of database management systems (DBMSs) has stimulated research to produce more efficient and easier-to-use systems. Greater efficiency is required in order to offset the inherent costs of operating a DBMS. One area of research directed at producing greater efficiency is in the development of database machines. Database machines use specially configured hardware, tailor-made software, and innovative techniques such as massive parallelism to support higher capacity and performance. Greater ease of use is necessary in order to ensure a wider distribution of use. The emergence of database systems using the relational model of data is an important development in this area.

One of the database machines of interest is the Multi-Backend Database System (MDBS). The idea of MDBS is to use general-purpose hardware and special-purpose software in a novel configuration to provide a backend database machine solution. The design and development of MDBS is an ongoing project [Ref. 1 and 2]. In this thesis, we will not examine the particular database machine solution to the efficiency problem. Rather the contribution of this thesis to the MDBS research is in the area of ease of use. We will determine

how the relational query language of SQL [Ref. 3] can be supported by the attribute-based query language of MDBS [Ref. 1]. In the next two sections, a brief review of the design goals of MDBS and of the development of relational query languages is presented. In the final section of the chapter, the organization of the thesis is discussed.

A. DESIGN GOALS FOR EFFICIENCY IN MDBS

As previously mentioned, research in database machines has been driven by the need to develop more efficient systems. Efforts have resulted in a variety of machines which include: CASSM [Ref. 4 and 5], RAP [Ref. 6], DBC [Ref. 7 and 8], DIRECT [Ref. 9], MDBS [Ref. 1 and 2], RDBM [Ref. 10], VERSO [Ref. 11], DBMAC [Ref. 12], and IDM [Ref. 13]. This is not a complete listing, but does illustrate the fact that no "best" architecture has been developed. Each of the machines listed are unique. This uniqueness makes classification impossible. However, while no true taxonomy of database machines exists, Strawser [Ref. 14] cites several design issues that can be used to categorize the systems. Three of these issues, processor structure, interconnection of the processor and the database store, and alternative physical organizations have particular relevance to the MDBS design. Within each of the issues there exist tradeoffs

that affect the performance of the various machines. What follows is a brief description of these three design issues, and the MDBS solutions.

Many database machines are organized with a single control processor and one or more slave processors. As in any such system, the control processor is a potential bottleneck. Some designs seek to alleviate this problem by having the control processor perform only administrative tasks, or by otherwise limiting its responsibility. At the other end of the spectrum, some machines permit the control processor to participate in query execution. Irrespective of the design chosen, throughput will be inversely proportional to the amount of work levied on the control processor. A goal of the MDBS design is to minimize the potential control processor bottleneck. The control processor performs a minimal set of functions, only those which are necessary to administer query execution.

Additional differentiation of processor structures can be made between homogeneous and heterogeneous multiprocessor organizations. Homogeneous organizations use a number of processors with identical functionality. This allows for a high degree of intra-query parallelism. The heterogeneous organization is characterized by a number of processors with specialized functionality, thus permitting inter-query

parallelism. MDBS uses a homogeneous multiprocessor organization, offering a high degree of intra-query parallelism. The software in the backend processors is identical, allowing easy expansion of the system by replicating the software when new backend processors are added. The backend processors operate in parallel. However, the backends also operate independently. Each backend has a separate scheduling mechanism, to make the optimum use of resources. Communication between the processors is via a broadcast bus, to minimize communication overhead.

There are two major categories to describe the interconnection of the processor and the database store. The first, direct interconnection, connects the processor directly to the database store. While this method has an advantage in that the processors never have to wait for data, it suffers in two respects. The processor must be able to work at speeds equivalent to the transfer rate of the secondary storage devices, and data sharing among processors is severely limited. The second major category is the hierarchical interconnection. This method, which is more prevalent, transfers data from the database store to RAM storage for access by the processors. In MDBS, each backend

processor has dedicated disk drives, eliminating contention between processors for the same device. Data is staged from the disk to buffers in the main memory.

Like other design issues, the motivation for seeking alternative physical organizations is to enhance performance. Two such designs are the data pool organization used in DBMAC [Ref. 12], and the V-Relation scheme used in VERSO [Ref. 11]. However, the gains realized from these organizations apply only to some operations. MDBS uses a clustering methodology to organize the database. Records in the database are divided into clusters based on attribute values. The clusters of the database are then spread across the backends, so that the advantages of parallelism are realized for all operations. In other words, for database access, MDBS attempts to achieve record-serial-cluster-parallel operations.

B. RELATIONAL QUERY LANGUAGE DEVELOPMENTS

Each successive generation of database languages has sought to make data manipulation more user-friendly. The idea is to remove from the user any responsibility for knowing the particularities of system structure. Early representations of databases, first the hierarchical model and then the network model, require the user to understand

the organization of the database in order to navigate through it for the purpose of storage, retrieval and update of the user data. The relational database approach attempts to present the user with an easy-to-understand tabular representation of the stored data which makes the storage, retrieval and update operations as simple as table manipulation.

Codd [Ref. 15] first proposed tuple relational calculus as a benchmark for evaluating data manipulation languages based on a relational model. The mathematical concept underlying the relational model is the set theoretic relation, which is a subset of the Cartesian product of a list of domains. A relation is any subset of the Cartesian product of one or more domains. Conceptually, a relation can be viewed as a simple, two-dimensional table that has several properties. First, the entries in the table are single-valued; neither repeating groups nor arrays are allowed. Secondly, the entries in any column are all of the same kind, that is each column has a domain of values that can appear in the column. Each column has a unique name and the order of the columns is immaterial. In the relational model columns are referred to as attributes.

The advantages inherent in the relational model are that no artificial constructs such as sets or pointers are

required, and that the data is represented in tabular (relational) form in a way that is familiar to the user. Operations on the data are specified logically or symbolically by relational algebra or calculus. This is of major importance in that while the data structure is predefined, the record relationships are not defined until they are used. Consequently, any relationship that can be expressed in relational algebra or calculus can be used. Among the advantages cited for relational DBMSs is increased productivity in applications development, due to the simplicity and flexibility of the model and the relational query languages.

The importance of the relational model in regards to this paper is not in its implementation, but rather the logical representation it offers to the user. This representation is developed through the use of relational query languages like SQL. SQL, earlier called SEQUEL, was first introduced by Chamberlin [Ref. 16] to be used with the relational model. It was another attempt to provide the user with an English-like language with which he could construct and manipulate his database. Developments and changes to the language grew out of IBM System R research [Ref. 3].

As pointed out by Hsiao [Ref. 17] the relational model suffers in its lack of solutions to the problems of database transformation and query translation. Conversely, any relational database may be transformed, in a straightforward way, into the attribute-based database used by MDBS. Therefore it is practical to think in terms of a relational database implemented on MDBS. Developing a relational query language interface to MDBS has several advantages. First, we provide an easy-to-use interface which afford the user the productivity increase claimed for relational query languages. Second, by choosing to implement the interface for SQL, the most widely used relational query language, we provide homogeneity for a wide community of database system users. Third, we identify those areas in MDBS where enhancements must be made in order to provide a full relational language capability.

C. ORGANIZATION OF THE THESIS

In Chapter 2 an overview of the organization of the multi-backend system is presented. Chapter 3 describes the MDBS query language. The SQL query language is described in Chapter 4. Chapter 5 explains the mappings from SQL to the MDBS query language. Chapter 6 offers recommendations for implementation, and Chapter 7 summarizes the conclusions obtained from the research experience.

II. ORGANIZATION OF THE MULTI-BACKEND SYSTEM

An understanding of the organization of the multi-backend database system is helpful in understanding some of the design considerations of the MDBS query language. Figure 1 is a representation of the MDBS hardware organization. The system is comprised of a controller and a number of backends, all general-purpose minicomputers. A broadcast bus connects the controller and the backends. Each backend has a dedicated number of disk drives.

The major design goals of MDBS are to allow the database to grow and the rate of requests to increase while maintaining good overall performance. To obtain these goals the multi-backend database system should have the following properties:

- (1) Throughput improvement is proportional to the multiplicity of backends;
- (2) Response time is inversely proportional to the multiplicity of backends;
- (3) The system is extensible for future growth and/or performance improvement;

These properties are obtained through various MDBS design considerations. MDBS seeks to minimize the potential of the control processor to be a bottleneck by minimizing the

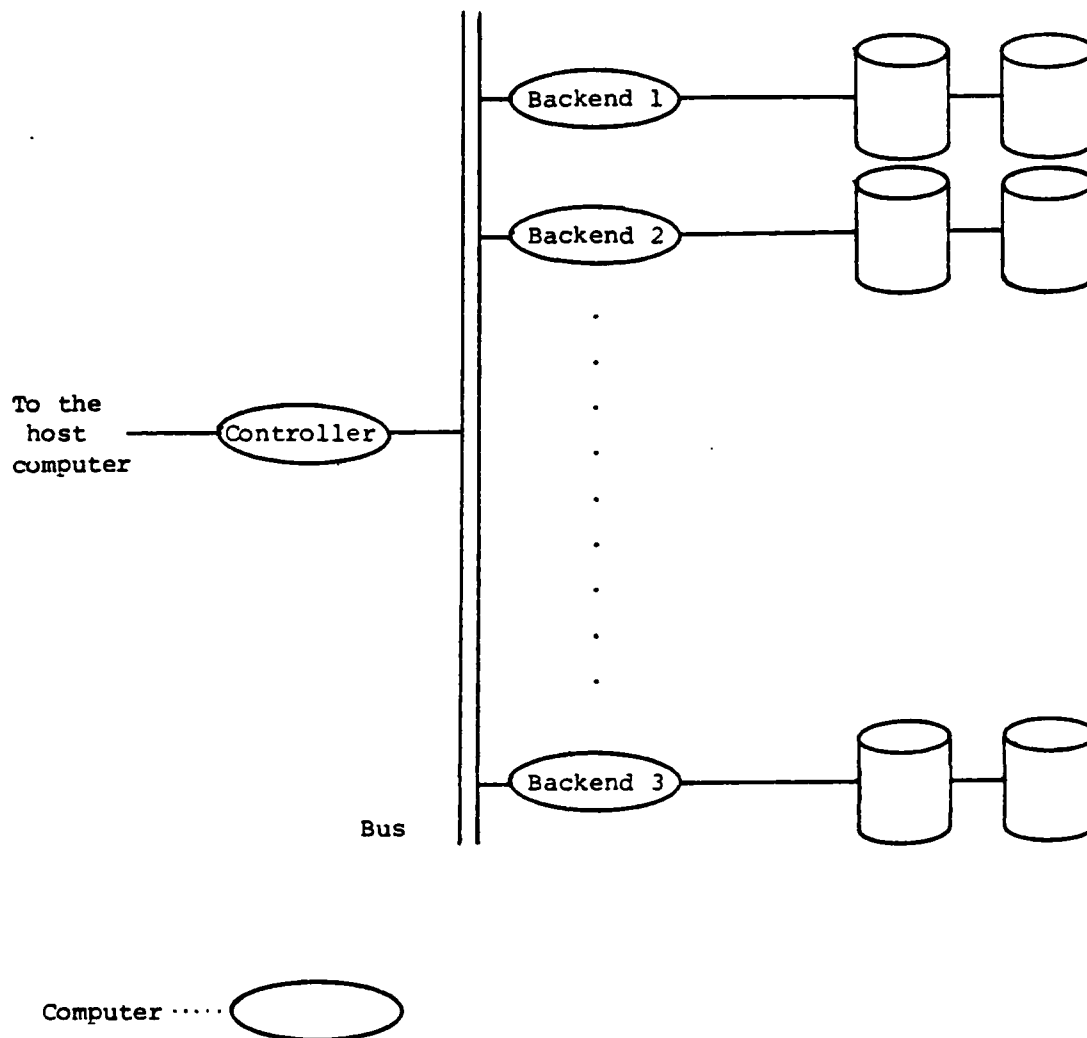


Figure 1. Overview of MDBS

controller functions. Accordingly, MDBS is viewed in terms of controller functions and backend functions, as depicted in Figure 2. Each backend is responsible for conducting its own operations, including queueing and scheduling of requests. Identical operating software is maintained at each backend. Expansion of the system is accomplished by replicating this software in additional backends.

The database is distributed across all the backends via the clustering mechanism, explained in the next section of this chapter. Requests from the controller are broadcasted to all the backends at the same time for processing. This allows for parallel processing of requests. Requests are queued at each backend. To permit continuous execution of requests each backend schedules request execution independently. The addition of more backends results in an increase in parallel processing of requests, which improves throughput and response time.

In the next three sections, descriptions of the attribute-based data model, the functions of the controller and the functions of the backend are presented.

A. THE ATTRIBUTE-BASED DATA MODEL

The data model used in MDBS is the attribute-based model developed by Hsiao and Harary [Ref. 17]. In their work they

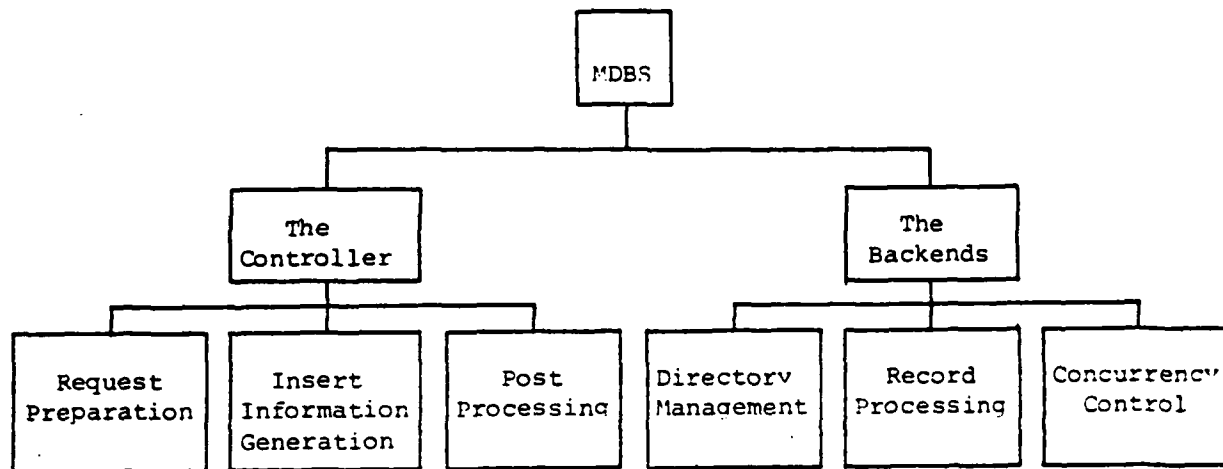


Figure 2. Functional Division of MDBS

use the set A to represent attributes and the set V to represent values. A record is then defined to be a subset of the Cartesian product $A \times V$, where each attribute has one and only one value. This way the record, R, is a set of ordered pairs of the form (an attribute, its value).

For each record R, a set of its attribute-value pairs which collectively characterize R is formed. These sets of a record collection form an index. These ordered pairs in the index are called the keywords. The index is used to identify a record or a set of records.

Following the keywords is the record body, which is a string of characters not used by MDBS for search purposes. An example of a record index without a following body is shown below.

(<FILE,employee>,<NAME,Smith>,<CITY,Monterey>,<RANK,3>)

The first attribute-value pair in all records of a file are the same, since it designates the file name. In the example above the file name is "employee".

In order to enhance the performance of the system, records are logically grouped into clusters. The clustering is determined by the attribute values and attribute value ranges in the records. In the example above, records could

be clustered on the NAME attribute, with all employees having a last name starting with the letter 'S' clustered together.

Keyword predicates are used in the data manipulation language for search and retrieval purposes. The keyword predicate has the form (attribute, relational operator, value). For example,

(SALARY > 2000)

is a simple greater-than predicate. A keyword is said to satisfy a predicate if the attribute of the keyword is identical to the attribute of the predicate and the relation specified by the relational operator of the predicate holds for the value of the keyword and the value of the predicate. For example, the keyword <RANK,4> satisfies the predicate (RANK < 6).

A conjunction is simply a conjunction of predicates, such as:

(SALARY > 10000) $\wedge$ (RANK = 3)

A record satisfies a query conjunction if the record contains keywords that satisfy every predicate in the conjunction. A query is a boolean expression of predicates, such as:

((DEPT = Sales) $\wedge$ (SALARY < 10000)) $\vee$
((DEPT = Sales) $\wedge$ (SALARY > 15000))

B. IMPLEMENTATION OF THE ATTRIBUTE-BASED MODEL IN MDBS

The indices of the attribute-based model are implemented in MDBS as descriptors. Descriptors are defined for designated directory attributes. The rules of definition require that the descriptors for a directory attribute form a partition over the domain of the attribute.

Clusters result from the partitioning of the database according to the descriptor definitions. A record belongs to the cluster defined by the set of descriptors which can be derived from the keywords of the record.

The clustering mechanism provides an ideal vehicle for distributing data across the backends of MDBS to take full advantage of parallelism. The records of a cluster are distributed track-at-a-time across all the backends. Therefore the work of query execution can be shared across the backends, with each backend processing the query against its portion of the relevant cluster(s). For a more detailed explanation of the clustering mechanism, readers are referred to [Ref. 1].

C. FUNCTIONS OF THE CONTROLLER

It is important to reiterate that a basic design consideration of MDBS is to minimize the functions of the controller. These functions are divided into three categories: request preparation, insert information generation, and post processing. The request preparation functions comprise the necessary operations performed on a request prior to its broadcast to the backends. These functions include parsing and syntax checking. Insert information generation functions are performed during the processing of an insert request in order to supply additional information needed by the backends. Post processing functions are performed after replies are returned from the backends. For example, these functions perform housekeeping duties on the separate responses to the single broadcast request, that is, the collection of the data prior to transmission to the host machine.

D. FUNCTIONS OF THE BACKEND

Functions within each backend are divided into three categories: directory management, record processing, and concurrency control. The directory management function is further divided into descriptor search, cluster search, address generation, and directory table maintenance. It is

responsible for searching through the descriptors and clusters to determine the disk addresses for the records to be accessed. The record processing functions include: record storage, record retrieval, record selection, and attribute value extraction of the retrieved records. Concurrency control is maintained by the locking of clusters to prevent conflicting access to the same clustered data.

Figure 3 is a representation of the operations performed on a user request. A request is submitted to the host, which converts it to the internal form of the MDBS environment. The controller parses the request and checks for syntax errors, then broadcasts the request to all of the backends. The work of descriptor search is shared among all backends. Each backend does its portion of descriptor search, and broadcasts its findings to all the other backends. When all descriptors have been identified, each backend independently performs cluster search. The appropriate records are then selected, values extracted and the results sent back to the controller. The controller collects the results from all the backends, performs any final aggregation required, and forwards the data to the host.

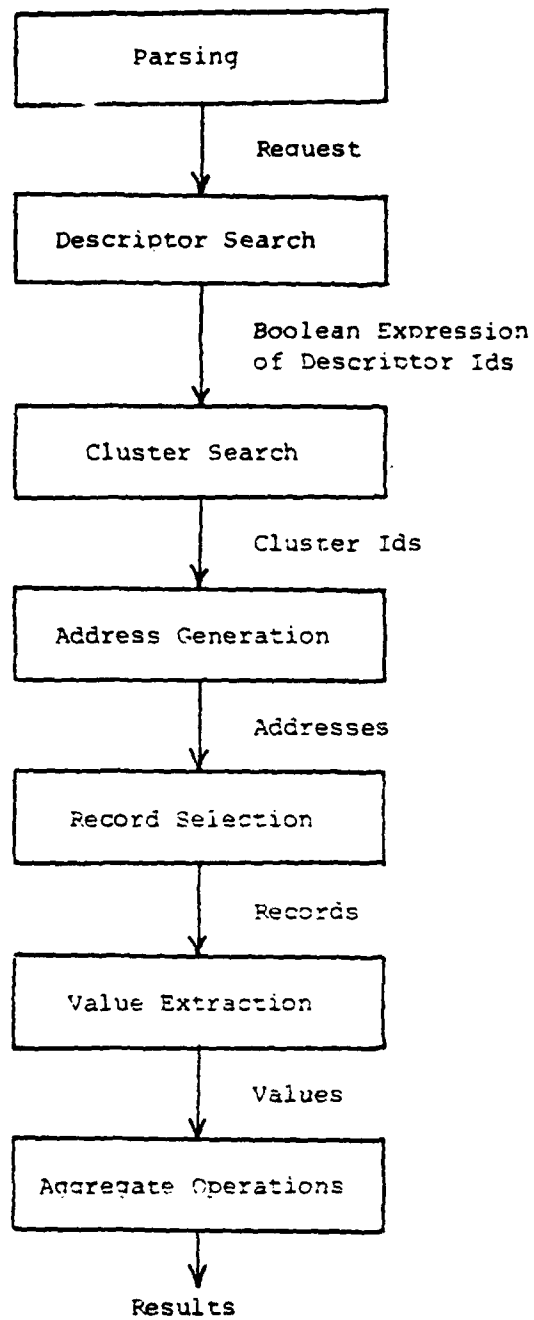


Figure 3. Request Flow in MDBS

III. THE MDBS QUERY LANGUAGE

The query language for MDBS is a non-procedural language in which queries are expressed in the disjunctive normal form. The language itself supports four different types of requests: retrieve, insert, delete, and update. Appendix A is a formal specification of the language. In the examples below, reserved words are capitalized and optional portions of queries are enclosed in brackets.

A. THE RETRIEVE REQUEST

The RETRIEVE is the most flexible of the operations on the database. It is the user's vehicle to query the database for information. Unlike the other three operations, the retrieve does not alter the contents of the database. Its syntax is:

RETRIEVE Query Target_list [BY attribute] [WITH pointer].

As shown above, the RETRIEVE request consists of five parts. The last two parts, those enclosed in square brackets, are optional. The operator RETRIEVE is a reserved word and indicates the type of request. The query is made up of predicates in the disjunctive normal form. The query defines the portion of the database which is to be retrieved. The target list is the list of attributes for

which values are to be extracted from the records which satisfy the query. The attribute value may be a value from the record, or an aggregate of values from multiple records. Five aggregate operators are supported in MDBS: AVG, COUNT, MAX, MIN, and SUM.

The BY clause performs an ordering on the data returned. For example, to RETRIEVE all the employees names ordered according to department, the following query can be used.

RETRIEVE(FILE = Employee) <NAME> BY DEPT

The WITH clause specifies whether pointers to the retrieved records must be returned to the user for later use in an update request.

Let us examine some examples of RETRIEVE requests.

Example 1. Retrieve the names of all employees who make less than \$10,000.

RETRIEVE((FILE = Employee) $\wedge$ (SALARY < 10000)) <NAME>

Example 2. Retrieve the average salary of employees who have a rank greater than 2, order by department.

RETRIEVE((FILE = Employee) $\wedge$ (RANK > 2))
<AVG(SALARY)> BY DEPT

B. THE INSERT REQUEST

The INSERT request is used to add records to the database. The syntax is:

INSERT Record

where record is the record to be inserted into the database.

An example of an INSERT request is:

```
INSERT(<FILE,Employee>,<NAME,Smith>,<SALARY,10000>)
```

This creates a record in the employee file for Smith and sets his salary at 10000.

C. THE DELETE REQUEST

The DELETE request is used to remove records from the database. The syntax is:

DELETE Query

where query is of the same form as that used in the RETRIEVE request. An example is:

```
DELETE((FILE = Employee) ^ (NAME = Smith))
```

This deletes all records in the employee file for employees named Smith.

D. THE UPDATE REQUEST

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

UPDATE Query Modifier

where the query specifies the particular records to be

modified and modifier indicates the type of modification that is to be performed. MDBS allows five types of modifications.

The TYPE-0 modification sets the new value of the attribute being modified to a constant. An example of a TYPE-0 modification is:

```
UPDATE((FILE = Employee) ^ (NAME = Smith))  
      <SALARY = 5000>
```

This sets the salary of all employees named Smith to 5000.

In the TYPE-I modification, the new value of the attribute is set to some function of the old value of the attribute in the record being modified. An example of a TYPE-I modification is:

```
UPDATE((FILE = Employee) ^ (NAME = Smith))  
      <SALARY = 2 * SALARY>
```

This doubles the salary of all employees named Smith.

The TYPE-II modification sets the new value of the attribute to some function of another attribute contained within the same record. Where a TYPE-I modification was a function of the same attribute, the TYPE-II modification looks at another attribute to derive a value. An example of

a TYPE-II modification is:

```
UPDATE((FILE = Employee) ^ (NAME = Smith))  
      <SALARY = 8 * RANK>
```

This makes all the salaries of employees with the last name Smith equal to eight times the value of their rank.

The TYPE-III modifier derives the new value of the attribute being modified from some function of another attribute value contained in another record which is identified by the query in the modifier. An example of a TYPE-III modification is:

```
UPDATE((FILE = Employee) ^ (NAME = Smith))<SALARY =  
      SALARY of (FILE = Positions) ^ (JOB = Manager)>
```

Here employees named Smith get their salary set to that of a manager's, as recorded in the Positions file.

The TYPE-IV modifier derives the new value of the attribute being modified from a function of another attribute value in another record identified by the pointer in the modifier. This requires a retrieval request first in order to obtain the value for the pointer. An example of a TYPE-IV modification is:

```
RETRIEVE((FILE = Employee) ^ (NAME = Jones)) with Pointer  
The retrieve request returns the value of a pointer, in this  
example, say, 2000. So we can then execute the following  
update request.
```

UPDATE((FILE = Employee) $\wedge$ (NAME = Smith))

<SALARY = SALARY of Pointer>

The effect of these two queries is that all employees with the name Smith have their salary set to that of Jones.

IV. THE RELATIONAL QUERY LANGUAGE, SQL

Data in the relational data model is depicted as a two-dimensional table. The relational query language, SQL attempts to exploit this representation. It does this by providing an English-like language that allows the user to list the attributes from a relation meeting the user's selection requirements. For a more complete description, the reader is referred to [Ref. 3 and 16].

Various implementations of SQL provide many functions and facilities beyond the basic SQL. The four basic constructs are: select, insert, delete, and update. However, in illustrating the use of the basic constructs, we include some other functions and facilities. In particular, some of the examples and constructs shown below are those implemented by the Oracle Corporation [Ref. 18] database management system.

A. THE SELECT REQUEST

The SELECT request is used for retrieval of data from the database. Its general form is as follows.

```
SELECT A ,...,A
FROM R
WHERE B  $\in$  b AND ... AND B  $\in$  b
```

where A and B are attributes found in the relation R, is a relational operator (such as >, <, =, >, <),, and b is a constant. In particular, $B \varnothing b$ is termed a predicate. Within the general guidelines above, SQL offers a great deal of latitude in the formation of the SELECT query. Let us look at each clause separately, the SELECT clause, the FROM line, and the WHERE line.

Instead of listing the attributes to be retrieved on the SELECT clause the user may request the return of the entire relation by using the wild card character, '*'. SQL also allows for the use of aggregate operators (such as AVG, SUM, MAX), arithmetic operators (such as +, -, /), and arithmetic functions (such as ROUND, TRUNC). Additionally, SQL permits the user to define the format for the retrieved data. These are only some of the basic variations permitted. Examples of these options follow:

Example 1. Retrieve all the attributes for all the employees. (Use of the wildcard.)

```
SELECT *  
FROM Employee
```

Example 2. Obtain the average salary of all the employees. (Use of an aggregate operator.)

```
SELECT AVG(Salary)
FROM Employee
```

Example 3. Obtain the total of the salary and commission for each employee. (Use of an arithmetic operator.)

```
SELECT Salary + Commission
FROM Employee
```

Example 4. Retrieve the salaries of all the employees, rounded to the nearest dollar. (Use of an arithmetic function.)

```
SELECT ROUND(Salary)
FROM Employee
```

Example 5. Retrieve the dates of hiring for all the employees, and format them to read month/day/year (ex. 09/24/50). (Use of a format option.)

```
SELECT TO_CHAR(Hiredate,'MM/DD/YY') Hiredate
FROM Employee
```

The FROM line identifies the relation or relations from which data is to be retrieved. A single relation is specified for simple retrievals. Two or more relations are specified for join operations.

An example of a simple SELECT on a single relation is as follows.

Example 6. Return the names of all the employees.

```
SELECT Name
FROM Employee
```

An example of a join, involving two relations in this case, is as follows.

Example 7. Return all the names and locations of the departments which have an employee named Smith.

```
SELECT Name, Location
FROM Employee, Department
WHERE Name = Smith
```

The WHERE line establishes the conditions on which the retrieval is to be made. Predicates are used to qualify the selection of tuples from the relations(s). Only those tuples which satisfy the predicates are selected. Like the

SELECT line it has many variations. These variations include: an attribute of the relation compared to some constant, the testing of an attribute for set membership, the use of boolean operators to create complex conditions, and the ability to nest additional SELECT clauses in order to extract values for comparison. The following are examples of some of these variations.

Example 8. Retrieve the names and salaries of all the employees that have a salary equal to 10000.
(Comparison of an attribute to a constant.)

```
SELECT Name, Salary
FROM Employee
WHERE Salary = 10000
```

Example 9. Obtain the names of the employees whose jobs are either a clerks, analysts, or managers.
(A test for set inclusion.)

```
SELECT Name
FROM Employee
WHERE Job IN (Clerk,Analyst,Manager)
```

Example 10. List the names of all the employees that have a salary equal to 10000 and are named Smith. (A logical AND operation.)

```
SELECT Name
FROM Employee
WHERE Salary = 10000 AND Name = Smith
```

Example 11. List the name and job of employees who have the same job as Smith. (A nested SELECT.)

```
SELECT Name, Job
FROM Employee
WHERE Job =
    (SELECT Job
     FROM Employee
     WHERE Name = Smith)
```

B. THE INSERT REQUEST

The INSERT request is used to create rows (tuples) in a relation (table) and has the general form:

```
INSERT INTO R
VALUES (V ,...,V)
```

where R is the relation name and V is a value. The order in which the data values are listed in the INSERT must correspond to the order of the columns in the table. An

example of an INSERT is as follows.

```
INSERT INTO Employee
VALUES (Smith,2,10000)
```

This example creates a new tuple within the employee relation. Assuming that the Employee relation has attributes name, rank, and salary, a new tuple is created with the name being Smith, the rank being 2, and the salary being 10000.

C. THE DELETE REQUEST

The DELETE removes a row (tuple) or rows (tuples) from a table (relation). It has the general form:

```
DELETE FROM R
WHERE B  $\phi$  b ,..., B  $\phi$  b
```

where R is the name of the relation, B is an attribute of the relation, ϕ is a relational operator, and b is a constant. The WHERE clause for the DELETE has the same options available that are in the SELECT. An example of a DELETE is as follows.

```
DELETE FROM Employee
WHERE Name = Smith
```

This deletes all rows from the Employee table where the name is equal to Smith.

D. THE UPDATE REQUEST

The UPDATE command changes the attribute values stored in the database. It has the general form:

```
UPDATE R
```

```
SET A = a ,..., A = a
```

```
WHERE B  $\neq$  b ,..., B  $\neq$  b
```

where R is the relation name, A is the attribute to be assigned a new value, a . The WHERE clause has options as previously discussed. An example of an UPDATE request is as follows.

```
UPDATE Employee
```

```
SET Salary = 20000
```

```
WHERE Name = Smith
```

This update results in all employees named Smith having their salaries set at 20000.

V. THE MAPPINGS FROM SQL TO THE MDBS QUERY LANGUAGE

The idea of a SQL interface to MDBS is to provide to the user a friendly interface. SQL was chosen as the query language because of its English-like syntax and the existence of a wide-spread community of SQL users. We must emphasize here that we are implementing an interface between the SQL users and MDBS. We are not adding functionality to MDBS.

The distinction between implementing an interface and adding functionality is important for the following reason. SQL is a relational query language. The primary operations of SQL are SELECT, UPDATE, INSERT, and DELETE. The special relational operations, projection and join, are included in SQL, as well as aggregate operations, ordering, and various set operations. SQL is usually supported by a relational database management system which implements all of these relational operations.

MDBS, however, is not based on the relational model. The data model of the MDBS machine is the attribute-based model. The attribute-based model is flexible, and can support relational data structures: relations, tuples, and attributes [Ref. 17]. However, the functionality of MDBS does not encompass all relational operations. The four primary

operations of the MDBS machine are RETRIEVE, UPDATE, INSERT, and DELETE. The aggregate operations are also supported. MDBS does not support the join and ordering operations. Nor does it support set operations.

From the discussion above, it is clear that the set of SQL operations to be included in our interface will be limited to those supported by the functionality of MDBS. The subset of SQL operations which can be supported by MDBS directly is formally specified in Appendix B. In the remainder of this chapter, we define the mappings from the subset of SQL which can be supported directly by the primary operations of MDBS. We present the mappings both in graphics and in text. In the next section we explain the graphic notations. The remaining sections of this chapter give the details of the mappings from SQL to the MDBS query language.

A. GRAPHIC NOTATION

We will show the mappings graphically, and also explain them in text. The graphic notation is illustrated in Figure 4. The general forms of the SQL and MDBS queries used here have been developed in Chapters IV and V. The mappings are represented by directional arrows, and symbols indicating the type of the mapping. We have identified two types of mappings: syntactic substitution and conversion.

Syntactic-substitution mappings require only simple substitution of syntactical terms. The symbol for this type of mapping is a square marked with the letter S. Figure 4 shows two examples of a syntactic substitution mapping. The first example maps the SQL SELECT term to the MDBS RETRIEVE term. The second example maps the SQL sel_expr_list to the MDBS target_list. This example illustrates that a syntactic substitution may be a direct copy of clauses from the SQL query to the MDBS query.

Conversion mappings combine a clause from a SQL query with information about the MDBS data structure to derive the clause of the MDBS query. The symbol for conversion mapping is a triangle marked with the letter C. In Figure 4, the mapping of the FROM and WHERE clauses of the SQL query into the query clause of the MDBS request is a conversion mapping.

In Section B, we present the overall structure of the mappings from SQL queries to MDBS queries. In Sections C, D, and E, we discuss individually the three conversion mappings identified in Section B.

B. MAPPING REQUESTS FROM SQL TO MDBS

In this section, we show the syntactic-substitution mappings for the general forms of the SQL SELECT, UPDATE,

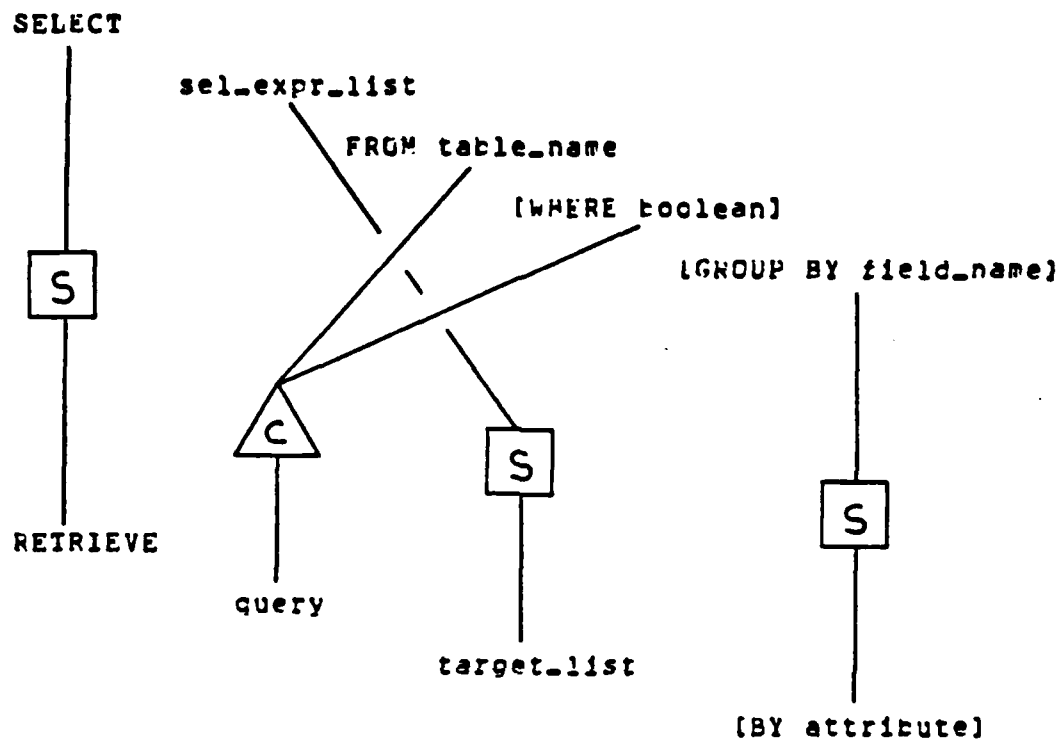


Figure 4. Mapping the SQL SELECT to the MDS RETRIEVE

INSERT, and DELETE requests into the MDBS RETRIEVE, UPDATE, INSERT, and DELETE requests, respectively. The conversion mappings are explained in detail in subsequent sections.

1. Mapping the SQL SELECT into the MDBS RETRIEVE

The first mapping is the SQL SELECT request to the MDBS RETRIEVE request. The SELECT query has the general form:

```
SELECT sel_expr_list FROM table_name
    [WHERE boolean]
    [GROUP BY field_name]
```

The RETRIEVE request has the general form:

```
RETRIEVE query target_list
    [BY attribute]
```

The SELECT to RETRIEVE mapping has been shown in Figure 4. The reserved word RETRIEVE is substituted for the reserved word SELECT. The sel_expr_list is a list of attributes that the user wishes to access from the database, and directly corresponds to the MDBS target_list. Consequently, it can simply be copied into the MDBS request. The "FROM table_name [WHERE boolean]" portion of the SQL request requires a conversion mapping into the "query" portion of the MDBS language. This conversion will be discussed in Section C. The reserved words 'GROUP BY' of SQL are directly translated into the MDBS reserved word, BY. The attribute upon which

the grouping is to take place is copied from the SQL query to the MDBS request.

2. Mapping the SQL INSERT into the MDBS INSERT

Figure 5 illustrates the mapping required for the insert requests. The general form for the SQL INSERT request is:

```
INSERT INTO table_name VALUES insert_spec
```

The MDBS INSERT request's form is:

```
INSERT record.
```

The reserved word INSERT is the same for the two requests. The remaining portion of the SQL request, 'INTO table_name VALUES insert_spec', requires a conversion mapping into the record portion of the MDBS query. This conversion will be explained in Section D.

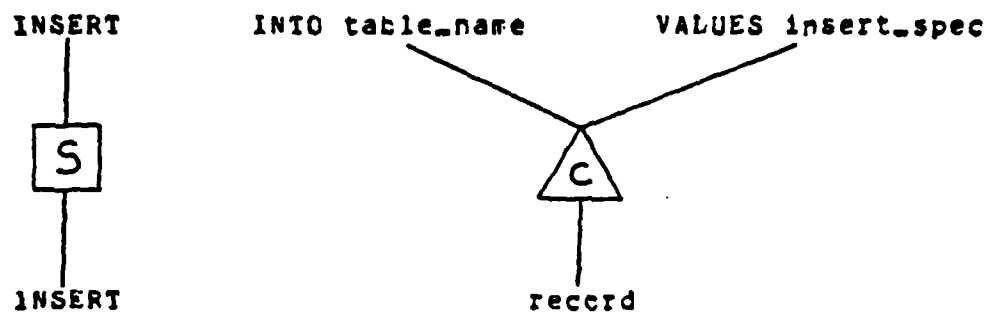


Figure 5. Mapping the SQL INSERT to the MDS INSERT.

3. Mapping the SQL DELETE into the MDBS DELETE

The mapping for the delete requests is shown in Figure 6. The delete request in SQL has the general form:

```
DELETE FROM table_name [WHERE boolean]
```

While in MBDS the general form is:

```
DELETE query.
```

The reserved word DELETE is common to both requests. The conversion of the "FROM table_name [WHERE boolean]" portion of the SQL request into the "query" portion of the MDBS request is the same as that required in the SELECT request, and will be discussed in Section C.

4. Mapping the SQL UPDATE into the MDBS UPDATE

Figure 7 depicts the mapping for the update request. The general form for the update request in SQL is:

```
UPDATE table_name set_clause_list  
[WHERE boolean]
```

In MDBS the form is:

```
UPDATE query modifier.
```

The SQL reserved word UPDATE is simply copied into MDBS. The "table_name [WHERE boolean]" conversion mapping is like that used in the SELECT and DELETE requests and will be explained in Section C. The set_clause_list of SQL requires a conversion mapping in order to match to the modifier

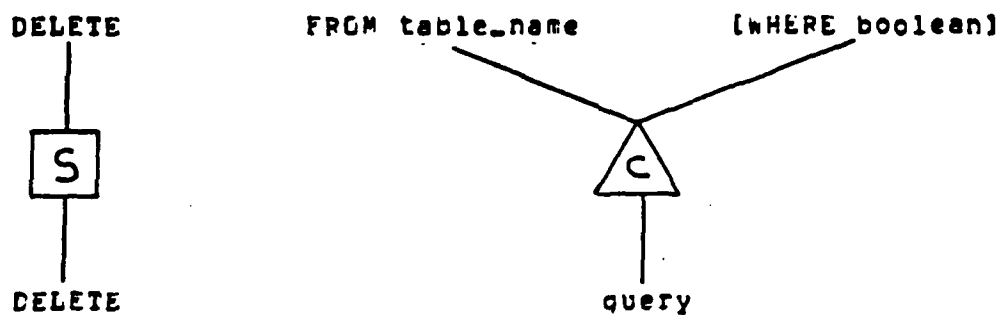


Figure 6. Mapping the SQL DELETE to the MDS DELETE.

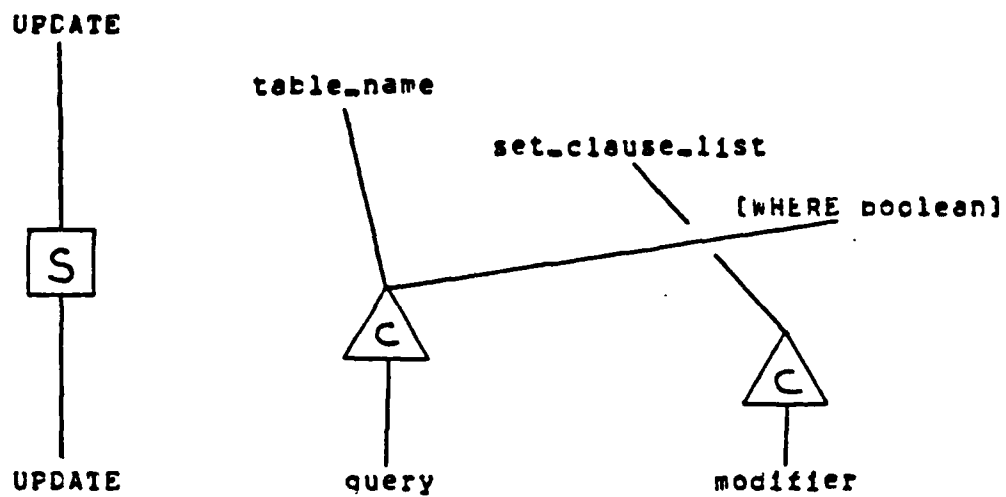


Figure 7. Mapping the SGL UPDATE to the MDS UPDATE.

portion of the MDBS request. That mapping is explained in section E.

C. THE CONVERSION MAPPING TO THE MDBS QUERY

The select, delete and update requests of SQL all have a "FROM table_name [WHERE boolean]" portion. In the update request it varies slightly in that the reserved word FROM is not used. However, the conversion required is essentially the same. This portion of the SQL request maps into the "query" portion of the MDBS retrieve, delete and update requests. However, due to the variety of forms and constructs available in SQL, a conversion is required to reconstruct this portion into an acceptable MDBS format.

As illustrated in Figure 8, much of the conversion requires only a simple mapping. The specification of the MDBS query requires that the first attribute-value relationship be "FILE = attribute", where the attribute is the name of a file. This is equivalent to the SQL, "FROM table_name".

In addition MDBS requires that queries be composed in the disjunctive normal form. SQL does not have this restriction. This is demonstrated in the examples below, where the SQL "[WHERE boolean]" clause is mapped into a disjunction of conjunctions in the MDBS request. To explain the conversions required to convert SQL's "boolean" into an

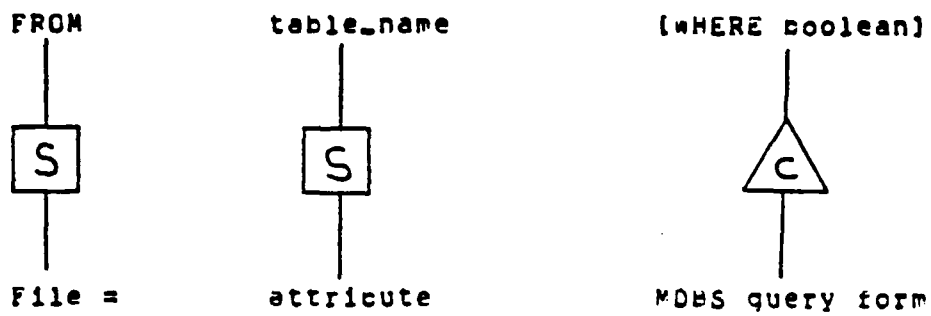


Figure 8. Mapping to the MDS query.

acceptable MDBS "query", examples will be used. In each case a SQL request will be shown, followed by the corresponding request in MDBS.

Example 1. Obtain the names of the employees that have a salary of 10000 and are clerks.

SELECT Name

FROM Emp

WHERE Sal = 10000 AND Job = Clerk

RETRIEVE

((File = Emp) $\wedge$

(Sal = 10000) $\wedge$ (Job = Clerk))

<Name>

Example 2. Obtain the names of employees who have
a salary between 5000 and 10000.

SELECT Name

FROM Emp

WHERE Sal BETWEEN 5000 AND 10000

RETRIEVE

((File = Emp) $\wedge$

((Sal $\geq$ 5000) $\wedge$ (Sal $\leq$ 10000)))

<Name>

Example 3. Obtain the names of employees who are
employed as a clerk, analyst or manager.

SELECT Name

FROM Emp

WHERE Job IN (Clerk, Analyst, Manager)

RETRIEVE

$((\text{File} = \text{Emp}) \wedge (\text{Job} = \text{Clerk})) \vee$

$((\text{File} = \text{Emp}) \wedge (\text{Job} = \text{Analyst})) \vee$

$((\text{File} = \text{Emp}) \wedge (\text{Job} = \text{Manager}))$

<Name>

As seen in example 3 above, the reconstruction of the SQL request into acceptable MDBS disjunctive normal form requires the identification of the file attribute in each predicate.

D. THE CONVERSION MAPPING TO THE MDBS RECORD

SQL's insert request uses "INTO table_name VALUES insert_spec" to identify the relation and attribute values that are to be inserted as a record. This corresponds to the "record" portion of the MDBS insert request. The MDBS record is a series of attribute-value pairs. The first pair is the file name (ex. <File,Emp>). This corresponds to SQL's "INTO table_name" which identifies the relation name. Figure 9 illustrates this mapping.

The "insert_spec" portion of the SQL insert request is a listing of the values to be inserted in the relation. The ordering of the values must be identical to the ordering of the attributes in the relation, and all attributes must have an assigned value. MDBS, on the other hand, represents a record as a list of attribute-value pairs. There is no requirement for ordering of the attribute-value pairs, as values are matched with attributes. Nor does MDBS require that values be assigned to all attributes. Instead MDBS assigns default values of zeros or spaces for integer and character attribute types.

In order to implement the SQL insert request, the MDBS record template information will have to be made available to the interface. The attribute names in the record template are ordered. The attribute names in the template

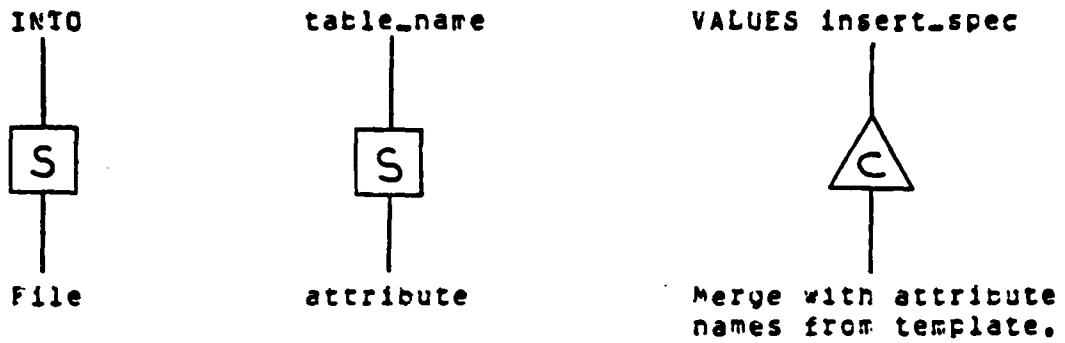


Figure 9. Mapping to the MDS record.

can then be matched to the values listed in the SQL insert request to form the attribute-value pairs of the MDBS record. For example, the following SQL insert request

```
INSERT INTO Emp VALUES Smith, Clerk, 10000
```

would be converted to read

```
INSERT (<File,Emp>, <Name,Smith>,  
      <Job,Clerk>, <Sal,10000>).
```

Alternatives for implementing this conversion will be further discussed in Chapter 6.

E. THE CONVERSION MAPPING INTO THE MDBS MODIFIER

The "set_clause_list" of the SQL update request has a direct correlation to the "modifier" of the MDBS update request. Figure 10 illustrates this mapping. SQL has constructs to represent the first four types of MDBS modifiers. The TYPE-0 modification sets the new value of the attribute being modified to a constant. The TYPE-I modification obtains the new value of an attribute being modified by setting it to some function of the old value. The TYPE-II modification sets the value of the attribute being modified to some function of another attribute contained within the same record. The TYPE-III modifier derives the value of the attribute being modified from some function of an attribute contained within another record. SQL has no construct which

SET field_name =

S

attribute_being_modified

expr

C

MDBS modifier form

Figure 10. Mapping to the MDDB modifier.

corresponds to a TYPE-IV modification, which derives the new value of the attribute being modified from a function of another attribute value in another record identified by the pointer in the modifier.

SQL offers a wide variety of constructs for its "expr" in the set_clause_list. In the examples below, we illustrate the correspondence between these constructs and the MDBS modifiers. The conversion required is a reordering or rewriting of these constructs into acceptable MDBS format. The conversion is much like that used in the query mapping of Section B. The following examples illustrate the conversions that are required. For simplicity, the examples are singular updates. The SQL request will be presented first, followed by the corresponding MDBS request.

Example 4. Set the salary of all employees named
Smith to 10000.

(Ex. MDBS Type-0 modification)

UPDATE Emp

SET Salary = 10000

WHERE Name = Smith

UPDATE ((File = Emp) ^

(Name = Smith))

<Salary = 10000>

Example 5. Double the salary of all employees
named Smith.

(Ex. MDBS Type-1 modification)

UPDATE Emp

SET Salary = 2 * Salary

WHERE Name = Smith

UPDATE ((File = Emp) $\wedge$

(Name = Smith))

<Salary = 2 * Salary>

Example 6. Set the salary of all employees
named Smith to eight times the value
of their rank.
(Ex. MDBS Type-II modification)

```
UPDATE Emp
      SET Salary = 8 * Rank
      WHERE Name = Smith
```

```
UPDATE ((File = Emp) ^
      (Name = Smith))
      <Salary = 8 * Rank>
```

Example 7. Set the salaries of the employees
named Smith to that of a manager's,
as recorded in the Positions file.
(Ex. MDBS Type-III modification)

```
UPDATE Emp
      SET Salary = (SELECT Salary
                    FROM Positions
                    WHERE Job = Manager)
      WHERE Name = Smith
```

```
UPDATE ((File = Emp) ^
      (Name = Smith))
      <Salary = Salary of (File = Positions) ^
      (Job = Manager)>
```

VI. RECOMMENDATIONS FOR IMPLEMENTATION

In Chapter V we demonstrated that a SQL-to-MDBS query language interface could be constructed for a subset of SQL. In this chapter, we will discuss the implementation issues. The first section deals with two areas of differences between the constructs of SQL and the MDBS query language. One is the MDBS requirement that queries be constructed in the disjunctive normal form. The other is the difference in the construct of the insert requests, as addressed in Chapter V, Section D.

In the second section we give suggestions for expanding the capabilities of the SQL/MDBS interface to support some SQL constructs that MDBS does not directly support. These are constructs which can be mapped from a single SQL request into a series of MDBS requests. The third section discusses extending MDBS to support the join and sort operations. The last section of this chapter discusses the use of program development tools to aid in the actual implementation.

A. SQL AND MDBS DIFFERENCES

In order to effectively support the SQL-to-MDBS interface, two differences in construct between the two languages have to be resolved. The first is the MDBS requirement that

all queries be written in the disjunctive normal form. The second is the form of the SQL insert request as compared to the MDBS insert request.

1. The Disjunctive Normal Form

MDBS requires the query portion of the retrieve, delete, and update requests to be written in disjunctive normal form. On the other hand most commercial versions of SQL do not place this restraint on the user. In order to support this capability, the interface will be required to translate the free-form logical SQL statements into the disjunctive normal form. In the very simple cases this is not an extraordinary burden. However, in any involved query the cost of translation could be expensive. For this reason, and to simplify construction of the interface, we believe that the user should be required to formulate his requests in the disjunctive normal form. This should not place a burden on the user since typically most requests are of a simple construction.

2. Differences in the Insert Request

The syntax of the insert request in SQL places a burden on the user to know the construction of the table into which he/she wishes to insert values. Each field must have an assigned value, and values must be listed in the order of the field_names in the table definition. MDBS, on the other

hand, specifies the record to be inserted as a list of attribute-value pairs. The attribute-value pair is a direct assignment of value to the indicated attribute. There is no constraint on the ordering of the pairs.

We recommend an enhancement for the SQL language interface, a new syntax for the insert request. The revised general form would be

```
INSERT INTO table_name VALUES insert_values.
```

The syntax for insert_values would be

```
insert_values := (field_name,insert_spec)
                | insert_values, (field_name,insert_spec)
```

This change in syntax brings the SQL insert command into line with the attribute-value pair syntax of the MDBS query language. More importantly it is believed that this change will improve user-friendliness.

B. EXPANDING THE FUNCTIONALITY OF THE INTERFACE

There are some basic SQL constructs which, while not directly supported by MDBS, can be mapped into a series of MDBS requests. The most important of these is the nested select construct. Additionally, commercial implementations of SQL offer a variety of features for manipulating and

editing data in result relations. We discuss below how the SQL-to-MDBS interface can be extended to provide these features.

1. Implicit Joins Through Nested Selects

SQL has the capability to nest select requests, as discussed in Chapter IV. MDBS does not have this capability. The SQL syntax dictates that the innermost nested select be evaluated first. Evaluation then proceeds outward. Translated into MDBS, this requires a succession of retrieve statements. The innermost select statement corresponds to the first retrieve request. The following is an example of a SQL request with a nested select, and a series of MDBS retrieve requests that obtain the same results.

Example. Obtain the names of the employees who have
a salary equal to that of a manager.

```
SELECT Name
FROM Emp
WHERE Sal = (SELECT Sal
              FROM Payroll
              WHERE Job = Manager)
```

```
RETRIEVE ((File = Payroll)  $\wedge$  (Job = Manager)) <Sal>
RETRIEVE ((File = Emp)  $\wedge$  (Sal = Sal)) <Name>
```

In the above example the Sal value obtained in the first MDBS request would be used as the Sal value in the second retrieve in order to obtain the Name.

In order to implement this capability in the interface we recommend that a pre-preprocessor be written that exclusively looks for nested selects. The pre-preprocessor finds the innermost select and sends it to the preprocessor. The value(s) obtained from the operation would then be inserted into the query portion of the next level select. This outward operation would continue until the entire

request had been executed. Utilization of the pre-preprocessor allows the preprocessor to operate on single select requests.

2. Formatting Options

SQL gives the user some options in the formatting of his/her output within the context of the select request. This includes creating new headings, indentations, and producing columnar/tabular outputs. The following example changes the column name based on the Sal attribute to a more readable heading, "Salary".

```
SELECT Name, Sal Salary
FROM Emp
WHERE Name = Smith
```

In order to make MDBS more user-friendly and useful in the area of report generation and formatting, we recommend that a post-processor be implemented as part of the interface. The post-processor would be responsible for performing the format and output options.

3. Arithmetic Operations and Functions

SQL affords the user the ability to specify arithmetic operations and functions on the values of the result relation. For example, the following select request creates a new column in the output called 'comm/sal', which is derived

from two existing attributes by dividing the comm attribute by the sal attribute.

```
SELECT Name, Comm/Sal, Comm, Sal
FROM Emp
WHERE Job = Salesman
```

The following is an example of a arithmetic function option in SQL.

```
SELECT Name, ROUND(Sal,2)
FROM Emp
WHERE Job = Salesman
```

This example rounds the value of Sal to two decimal places. These and similar operations, can be implemented in a post-processor.

C. JOIN AND SORT OPERATIONS

SQL and the relational data model support join and sort operations. Currently MDBS does not support either. Implementation of the nested select, as discussed in the previous section, would enable MDBS to support implicit joins, i.e., nested select requests. As MDBS is still in development, further research is required into the feasibility and desirability of implementing these operations on MDBS. The

considerations of costs versus the additional capability that would be provided by such an implementation is outside the scope of this paper. However, if implemented, the effort to include the required SQL-to-MDBS translation in the interface would be minimal.

D. TOOLS FOR ACTUAL IMPLEMENTATION

We recommend that the actual implementation of the interface be done using Yacc [Ref. 19] and Lex [Ref. 20], programming tools developed at Bell Laboratories. They can be used to produce an interpreter which accepts SQL requests and outputs the translated MDBS request.

Lex is a lexical analyzer generator, designed for lexical processing of character input streams. The user supplies the specifications for character string matching, Lex then produces a program in the programming language C, which recognizes regular expressions. Lex is generally used with Yacc to recognize and supply tokens.

Yacc, an acronym for Yet Another Compiler-Compiler, is a general tool for imposing structure on the input to a computer program. The user prepares a specification of the input process, i.e., rules and actions. Yacc then generates a program of functions to control the input process. Yacc calls the lexical analyzer (Lex) to supply tokens, and then

parses the supplied tokens according to the production rules.

Utilization of Yacc and Lex has several advantages. First, the MDBS query parser was created using these tools. Therefore in the event of any required scanner/parser to scanner/parser communications, both would be operating in similar environments. Second, both are written in C, which improves their transportability. Third, Yacc and Lex are both well documented and are relatively easy to use.

VII. CONCLUSION

MDBS uses the attribute-based data model. Records are composed of ordered pairs of the form (an attribute, its value). Descriptors, or indices, are defined for selected directory attributes. These descriptors are used to partition the database into clusters. The clusters are distributed across the backends to take full advantage of parallel execution of requests.

The MDBS user accesses the database using a simple, non-procedural query language. The language supports four different types of requests: retrieve, insert, delete, and update. The retrieve query is used to access, but not alter, the contents of the database. The insert and delete requests are used to add or remove records in the database. The update request modifies existing records of the database.

SQL is a relational query language, designed for use with relational databases. Like the MDBS query language, it has four different types of requests: select, insert, delete, and update. Like the MDBS retrieve request, the select accesses, but does not alter, the contents of the database. The SQL insert, delete, and update requests perform operations similar to those of their MDBS counterparts.

However, unlike the MDBS query language, SQL offers a variety of options in the syntax of its requests. This variety enables SQL requests to be constructed with varying degrees of logical and syntactical complexity.

In this thesis, we have identified the direct mappings from SQL queries into MDBS queries. These mappings can be directly supported in the SQL-to-MDBS interface. We have also identified those SQL constructs which have no direct mapping, but can be converted into a sequence of MDBS queries. Enhancements to the interface are proposed to support these indirect mappings. Lastly, we have identified those SQL constructs for which no mapping exists. To support these mappings, the functionality of MDBS must be augmented. Let us discuss each of these cases, identifying the contributions of this thesis and directions for further research.

A. THE DIRECT MAPPINGS

Some SQL queries can be directly mapped into MDBS requests. The retrieve, insert, delete, and update requests of MDBS have a direct functional correspondence to the SQL select, insert, delete, and update requests, respectively. There are three exceptions which require a degree of insight in order to perform a mapping. These are the mapping of the

"FROM table_name [WHERE boolean]" portion of SQL into the "query" portion of MDBS, the mapping of the SQL "INTO table_name VALUES insert_spec" into the MDBS "record", and the mapping of the SQL "set_clause_list" into the MDBS "modifier".

MDBS requires that the query portion of the request be written in disjunctive normal form. SQL, on the other hand, allows for free formatting of its logical constructs. To convert the SQL "FROM table_name [WHERE boolean]" construct into acceptable MDBS "query" format requires translating the options contained in the SQL "boolean" into MDBS disjunctive normal form. The complexity of this translation is $O(n^2)$, where n is the number of predicates in the boolean expression. In order to limit the overhead of this translation, we recommend that the SQL-to-MDBS interface require the user to construct SQL qualifications in disjunctive normal form.

The SQL insert request uses "INTO table_name VALUES insert_spec" to identify the relation and to list the values to be inserted. This list of values must correspond in order and type to the constructed relation. MDBS, on the other hand, uses attribute-value pairs for insert parameters. One solution to this conversion is to have the SQL-to-MDBS interface provide to the user the MDBS record template for assignment of values. Another approach, which we recommend,

is to alter the syntax of SQL's insert request to make it correspond to the attribute-value pair syntax of the MDBS "record". This eliminates the requirement that the user know the exact structure of the relation definition.

SQL's "set_clause_list" and MDBS's "modifier" are used to identify the attributes to be changed as a result of an update request. In addition, they specify the type of update. With the exception of the TYPE-IV modification in MDBS, which uses a pointer, there is a direct correspondence between the two languages in the syntax of their update requests. The only conversion required is formatting the "set_clause_list" into one of the acceptable MDBS "modifier" types.

These direct mappings have been fully explained in this thesis. Further research will involve implementing the interface. For implementation a lexical scanner and an interpreter could be constructed using the Yacc and Lex programming tools.

B. ENHANCEMENTS TO SUPPORT FURTHER MAPPINGS

There are several constructs which cannot be supported by direct mappings, but can be supported by an enhanced interface. The first of these is the implicit join operation, implemented in SQL by the nested SELECT. A re-preprocessor can be constructed to convert the nested SELECTs into a series of MDBS queries, and to control the iterative execution required.

Several options are available in commercial versions of SQL which are not supported in MDBS, such as arithmetic operations and functions, and output formatting. In order to implement these features a post-processor could be constructed. Further research will be required to design and analyze these pre- and post-processor functions.

C. OPERATIONS FOR WHICH NO MAPPING EXISTS

The SQL options that cannot be supported by MDBS are related to the relational join operation and to the sorting capability commonly found in relational systems. MDBS, which is not a relational system but an attribute-based system, does not support either the join or the sort operation.

In order to provide a fully-functional relational interface to MDBS, some provision must be made to implement these operations. There are two choices. First, the join and

sort operations could be implemented in MDBS. Second, these operations could be preformed by additional software running on the host. Further research will be required to identify the costs and the tradeoffs of these two alternatives.

APPENDIX A: FORMAL SPECIFICATION OF DML FOR ATTRIBUTE-BASED LANGUAGE

The following is the BNF for the attribute-based data manipulation language developed by Hsiao and Menon [Ref]. Square brackets [] are used to indicate optional constructs.

Predicate	:= attribute rel_op value
attribute	:= char_string
attribute_being_modified	:= attribute
base_attribute	:= attribute
value	:= string number float
Conjunct	:= (Predicate) (Conjunct / Predicate)
Query	:= Conjunct Query / Conjunct
Stat	:= AVG MAX MIN SUM COUNT
list_el	:= Stat (attribute)
list	:= attribute list_el list, attribute list, list_el
Target_list	:= (list)
Attrib_val_pair	:= <attribute, value>
Half_record	:= Attrib_val_pair Half_record, Attrib_val_pair
Record	:= (Half_record)

Pointer	:= number
Modifier	:= type-0 type-I type-II type-III type-IV
type-0	:= <attribute_being_modified = value>
type-I	:= <attribute_being_modified = expr1>
type-II	:= <attribute_being_modified = expr2>
type-III	:= <attribute_being_modified = expr2 of Query>
type-IV	:= <attribute_being_modified = expr2 of Pointer>
Request	:= Insert Delete Update Retrieve
Insert	:= INSERT Record
Delete	:= DELETE Query
Update	:= UPDATE Query Modifier
Retrieve	:= RETRIEVE Query Target_list [BY Attribute] [WITH Pointer]
uc_letter	:= A B C ... Z
string	:= uc_letter String uc_letter
lc_letter	:= a b c ... z

char_string	:= uc_letter char_string lc_letter
digit	:= 0 1 2 ... 9
number	:= digit digit number
float	:= number.number
add_op	:= + -
mult_op	:= * /
expr1	:= arith_term1 expr1 add_op arith_term1
arith_term1	:= arith_factor1 arith_term1 mult_op arith_factor1
arith_factor1	:= attribute_being_modified number
expr2	:= arith_term2 expr2 add_op arith_term2
arith_term2	:= arith_factor2 arith_term2 mult_op arith_factor2
arith_factor2	:= base_attribute number

APPENDIX B: FORMAL SPECIFICATION OF DML FOR SQL MAPPING

The following is the BNF for the SQL query language to MDBS query language mapping. Square brackets [] are used to indicate optional constructs.

```
dml_statement      := selection;
                    | insertion;
                    | deletion;
                    | update;

insertion          := INSERT INTO table_name VALUES
                    insert_spec

insert_spec        := selection
                    | literal

deletion           := DELETE FROM table_name
                    [ where_clause ]

update            := UPDATE table_name set_clause_list
                    [ where_clause ]

where_clause       := WHERE boolean

set_clause_list    := set_clause
                    | set_clause_list , set_clause

set_clause         := SET field_name = expr
                    | SET field_name = ( selection )

selection          := query_block
                    | ( selection )

query_block        := select_clause FROM table_name
                    [ WHERE boolean ]
                    [ GROUP BY attribute ]

select_clause      := SELECT sel_expr_list
                    | SELECT *
```

```

sel_expr_list      := sel_expr
                    | sel_expr_list , sel_expr

sel_expr           := field_name
                    | stat ( field_name )

boolean            := boolean_term
                    | boolean OR boolean_term

boolean_term       := boolean_factor
                    | boolean_term AND boolean_factor

boolean_factor     := [ NOT ] boolean_primary

predicate          := attribute comparison value
                    | attribute BETWEEN value AND value
                    | attribute NOT BETWEEN value AND
                      value
                    | attribute comparison table_spec
                    | table_spec comparison
                      full_table_spec

full_table_spec    := table_spec
                    | value

table_spec         := query_block
                    | ( selection )
                    | ( literal )

expr               := arith_term
                    | expr add_op arith_term

arith_term         := arith_factor
                    | arith_term mult_op arith_factor

arith_factor       := primary

primary            := value
                    | ( expr )

comparison         := rel_op
                    | IN
                    | NOT IN

rel_op             := = | <> | < | > | <= | >=

add_op             := + | -

```

mult_op	:=	* /
stat	:=	AVG MAX MIN SUM COUNT
literal	:=	lit_tuple_list lit_tuple
lit_tuple_list	:=	lit_tuple lit_tuple_list , lit_tuple
lit_tuple	:=	value
table_name	:=	attribute
field_name	:=	attribute
attribute	:=	char_string
value	:=	string number float
char_string	:=	uc_letter char_string lc_letter
string	:=	uc_letter string uc_letter
uc_letter	:=	A B C ... Z
lc_letter	:=	a b c ... z
number	:=	digit number digit
float	:=	number . number
digit	:=	0 1 2 ... 9

LIST OF REFERENCES

1. Menon, M. J., Hsiao, D. K., "Design and Analysis of a Multi-Backend Database System for Performance Improvement, Functionality Expansion and Capacity Growth (Part I)", Technical Report OSU-CISRC-TR-81-71, The Ohio State University, 1981
2. He, X., and others, "The Implementation of a Multi-Backend Database System (MDBS): Part II - The Design of a Prototype MDBS", Advanced Database Machine Architectures, Prentice-Hall, 1983, pp. 327-385
3. Astrahan, M. M., and others, "System R: Relational Approach to Database Management", Transactions on Database Systems, Volume 1, Number 2, Jun 1976, pp. 97-137
4. Su, S. Y. W., "Cellular-Logic Devices: Concepts and Applications", Computer, Mar 1979, pp. 11-25
5. Su, S. Y. W., "The Architectural Features and Implementation Techniques of the Multicell CASSM", IEEE Transactions on Computers, Vol C-28, No. 6, Jun 1979, pp. 430-445
6. Ozkaharahan, E. A., Schuster, S. A., and Smith, K. C., "RAP -- An Associative Processor for Data Base Management", Proceedings of the National Computer Conference, 1975, pp. 379-387
7. Bannerjee, J., Hsiao, D. K., and Baum, R. I., "Concepts and Capabilities of a Database Computer", ACM Transactions on Database Systems, Vol. 3, No. 4, Dec 1978, pp. 347-384
8. Bannerjee, J., Hsiao, D. K., and Krisnamurthi, K., "DBC - A Database Computer for Very Large Databases", IEEE Transactions on Computers, Vol. C-28, No. 6, Jun 1979, pp. 414-429

9. DeWitt, D. J., "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems", IEEE Transactions on Computers, Vol C-28, No. 6, Jun 1979, pp. 395-406
10. Schweppe, H., and others, "RDBM: A Dedicated Multiprocessor System for Data Base Management", Advanced Database Machine Architectures, Prentice-Hall, 1983, pp. 36-86
11. Bancilhon, F., and others, "VERSO: A Relational Backend Database Machine", Advanced Database Machine Architectures, Prentice-Hall, 1983, pp. 1-18
12. Missikoff, M. and Terranova, M., "The Architecture of a Relational Database Computer Known as DBMAC", Advanced Database Machine Architectures, Prentice-Hall, 1983, pp. 87-108
13. Britton Lee, Inc., Intelligent Database Machine Product Description, 1983
14. Strawser, P. R., "A Methodology for Benchmarking Relational Database Machines", Ph.D. Dissertation, The Ohio State University, 1983
15. Codd, E. F., "Relational Completeness of Data Base Sublanguages", Data Base Systems, Prentice-Hall, 1972, pp. 65-98
16. Chamberlin, D. D., and Boyce, R. F., "SEQUEL: A Structured English Query Language", Proc. ACM SIGFIDET Workshop, Ann Arbor, Mich., May 1974, pp. 249-264
17. Hsiao, D. K., and Harary, F., "A Formal System for Information Retrieval from Files", Communications of the ACM, Vol. 13, No. 2, Feb 1970, pp. 67-73
18. Oracle, Relational Software Incorporated, Menlo Park, California, Mar 1983
19. Johnson, S. C., "Yacc : Yet Another Compiler-Compiler", Bell Laboratories, Murray Hill, N. J., Jul 1978

20. Lesk, M. E. and Schmidt, E., "Lex - A Lexical Analyzer Generator", Bell Laboratories, Murray Hill, N. J., Jul 1978

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2. Library, Code 0142 Naval Postgraduate School Monterey, California 93943	2
3. Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
4. Curricular Officer, Code 37 Computer Technology Naval Postgraduate School Monterey, California 93943	1
5. Dr. D. K. Hsiao, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943	1
6. Dr. P. R. Strawser, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943	1
7. HQ COMNAVSECGRU ATTN: LCDR Curtis Ryder, G30D 3801 Nebraska Avenue Washington, DC 20390	1
8. Commanding Officer ATTN: LT Griffin N. Macy Naval Security Group Activity Northwest Chesapeake, Virginia 23322	1