AN ABSTRACT OF THE THESIS OF

Patricia M. Kalvin for the degree of Master of Science in Computer Science presented on March 6, 1984. Title: A Dataflow Mechanism for Supporting Query Optimization

Abstract approved: Redacted for Privacy

Michael J. Freiling

The objective of this thesis is to develop a tool which can be used to implement query processing algorithms produced by a query optimizer. The tool should have the following properties:

(1) it should support a description of the query solution in a dataflow-like language,

(2) it should support data retrieval functions which are independent of the rest of the system,

(3) it should allow file access to be treated as a virtual operator,

(4) it should be able to run on today's serial architectures, yet have the capability to expand to future parallel systems.

The algorithm for processing a query can be described easily and naturally using a dataflow-like language. Because solutions to queries involve streams of data, i.e.

each file access can be visualized as an operator producing a stream of data, dataflow languages lend themselves to easily describing query solutions.

By making data retrieval functions independent of the rest of the system, new technologies in data storage can easily be added to an existing system. The system can also be more responsive to user needs by allowing file organization to be changed without having to recompile the entire system.

Virtual file access allows the underlying file organization to be transparent to the query optimizer. This means that new file organizations can be handled by the optimizer without having to restructure the optimization strategy.

The constraint of a serial processor is present because this allows problems that benefit from a dataflow approach to be solved using that approach even though the only processor available is a serial processor.

A DATAFLOW MECHANISM FOR SUPPORTING QUERY OPTIMIZATION

by

Patricia M. Kalvin

A THESIS

submitted to

Oregon State Univeristy

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed March 6, 1984

Commencement June 1984

APPROVED:

Redacted for Privacy

Assistant Professor of Computer Science in charge of major

Redacted for Privacy

Chairman of the Department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis is presented March 6, 1984

Typed by Patricia Kalvin for Patricia M. Kalvin

TABLE OF CONTENTS

1. INTRODUCTION..1

   1.1 An Overview....................................1
   1.2 Back-end Processing............................13
   1.3 The Dataflow Approach..........................26

2. A DATAFLOW FRAMEWORK FOR ACCESS PATH DEFINITION.30

   2.1 Overview.......................................30
   2.2 Query Processing Framework.....................32
   2.3 Dataflow Access Modules........................35
   2.4 Data Flow Access Language......................42

3. THE FILE ACCESS MACHINE............................46

   3.1 The File Access Machine........................46
   3.2 Run Time Environment...........................47
   3.3 The File Access Language.......................52
   3.4 File Access Language Operation.................54
   3.4.1 Control Structures...........................54
   3.4.2 Arithmetic Statements........................55
   3.4.3 Test Operators...............................55
   3.4.4 File Primitives..............................56
   3.4.5 Data Movement................................57

4. DATAFLOW ACCESS LANGUAGE IMPLEMENTATION.........58

   4.1 Code Expander..................................60
   4.2 Dataflow Meta-Language.........................63
   4.2.1 Do Operations................................65
   4.2.2 Write Output.................................67
   4.2.3 Get Next Input Record........................70
   4.2.4 Get Input Field..............................71
   4.2.5 Get Next Work Record.........................73
   4.2.6 Get Work Field...............................73
   4.2.7 File Initialization..........................73
   4.2.8 Close..74
   4.2.9 Do Final Operations..........................75
   4.3 Assembling the Code............................76
   4.4 The Code Generator in Action...................77

5. CONCLUSION...89

   Bibliography.......................................91
   Appendix A...93
   Appendix B...102
   Appendix C...103
   Appendix D...106

LIST OF FIGURES

Figure		Page
1	DBMS Architecture	3
2	Relations	4
3	Result of a Join operation	6
4	Query Solution	15
5	Yao's seven classes	18
6	SEQUEL algorithm in Yao's scheme	19
7	A dataflow graph	33
8	Overview of Query Processing in Osiris	34
9	Sample query	36
10	Basic Module Structure	37
11	Sample Query	39
12	Sample Schema	39
13	Data Flow Access Graph for sample query	40
14	Module Descriptor for Module AM-1	41
15	Data Flow Access Language Meta-Commands	43
16	Algorithm for processing an access module	43
17	Template A	44
18	Operators for File Access Machine	47
19	Symbol Table	48
20	Private Registers for File Access Machine	48
21	Public Registers	50
22	Outline of File Access Machine	52
23	Partial File Access Language Program	54
24	Use of Module Descriptor	60

25 Data Flow Access Machine Variables 62

26 Template A 64

27 Data Flow Access Language Meta-Commands 64

28 Generated Code --- DOP 67

29 Code to check output buffer 69

30 File Access Machine code generated by WRO 71

31 Code generated by Get Next Input Record 72

32 Code generated by Get Input Field 72

33 Final Operations Stack contents 75

34 Code generated by Do Final Operations 76

35 Module Description for Module AM-1 78

36 Template A 79

37 Expanded code file 81

38 Code to get first operand 82

39 DOP expanded for Module 1 83

40 Code to check output buffer 85

41 File Access Machine code generated by WRO 86

42 Main loop for Template A 87

43 Expanded meta-commands for access module AM-1 88

A DATAFLOW MECHANISM FOR SUPPORTING QUERY OPTIMIZATION

INTRODUCTION

This thesis presents a tool which can be used to implement query processing algorithms produced by a query optimizer. Specifically, the tool will allow algorithms which are used for query optimization to be designed independently of the access methods needed to retrieve data. The remainder of this chapter will present an overview of current database management systems and the relationship between query optimization and data retrieval in these systems.

Chapter Two will present a dataflow framework which can be used in access path definition. This chapter will also present the Data Flow Access Language which is used to describe query processing strategies. Chapter Three will discuss the File Access Machine, which implements the framework developed in Chapter Two. Chapter Four will describe how the Data Flow Access Language is expanded into executable code which runs on the File Access Machine. This chapter will also present an example.

1.1. AN OVERVIEW

A database management system (DBMS) is a set of programs that allow one or more users to modify data stored in a computer [DAT 77]. The advantages of using a DBMS

are:

(1) the users can deal with data in abstract terms;

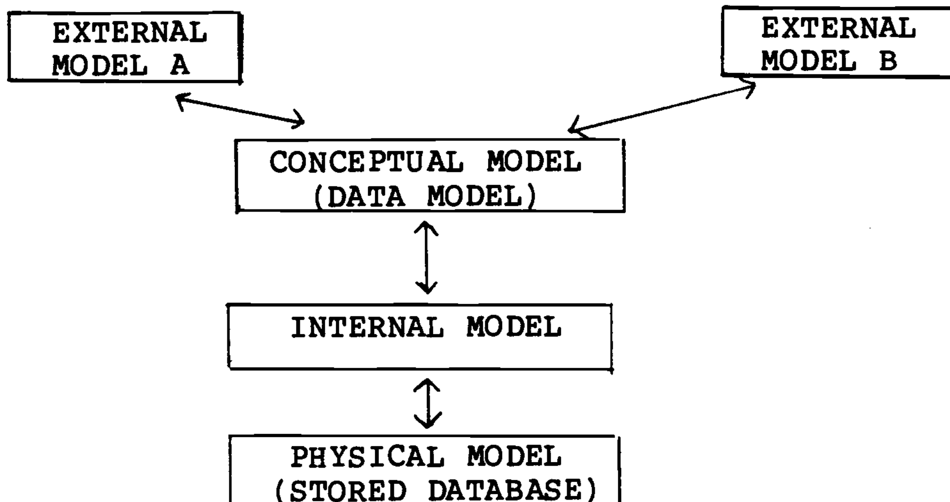
(2) data modification can be controlled so that users cannot enter inconsistent data into the database;

(3) the DBMS can control who is allowed access to what data.

A DBMS architecture is shown in Figure 1. The DBMS has several levels of abstraction. These levels are the external level, the conceptual (or data) level, the internal level and the physical level.

The most abstract level is called the external level. At this level, the external level defines the database differently for each user. The database that each user sees is called a <u>view</u>.

For example, a database for a large company contains payroll information, inventory information, and sales information. An employee in the payroll department sees only the payroll information, and an employee in the sales department sees only the sales information, while an employee in the accounting department may see all the information. Each of these employees sees a "view" of the database. Each employee may think of the database he or she works with as a separate database, but the DBMS allows



DBMS Architecture
FIGURE 1

the data to be stored and modified centrally.

The next level of abstraction after the external level is the conceptual level. This level is also referred to as the data level. This level contains a description of all the data in the database. The description of the data is called the data model. The data model describes relationships in the database and it describes restrictions to be applied when data is modified, or added to the database. For example, assume we are looking at the database for the mythical XYZ company. The payroll department requires that each employee-id be unique, therefore the data model shows that any time an employee-id is added or modified, a check must be performed to

ensure that there are no duplicate employee-ids in the database.

One data model that is used frequently is the relational data model. In the relational model a database is defined as a set of two-dimensional tables called relations [COD 70]. Each relation is a two-dimensional table with n-columns and m-rows. Each row in the table is called a _tuple_. Each column in the relation is called an _attribute_. The set of values that an attribute can be assigned is called the _domain_ of the attribute. Figure 2 shows two relations used in the data model for the XYZ company.

ID	NAME	LOC
2	R&D	OR
6	ADMIN	OR

DEPARTMENT RELATION

NAME	ID	DNO	SALARY
BROWN	040	6	50,500
SMITH	002	6	30,300
JONES	020	2	15,000

EMPLOYEE RELATION

Relations
FIGURE 2

The Department relation has three attributes, DEPT-NAME, DEPT-ID and DEPT-LOC. (Often when referring to attribute names in a relation, the relation name is used as the prefix.) The second relation is the Employee relation. This relation has the attributes: EMP-NAME, EMP-ID, EMP-DNO, and EMP-SALARY. There are three tuples in the

Employee relation and two tuples in the Department relation. It is important to note that the attribute names have no semantic meaning. Thus the same attribute name used in two different relations may refer to attributes that have no connection. In this case, the attribute name ID stands for the employee id in the Employee relation, and for department number in the Department relation.

The relational model includes a relational algebra [COD 70] consisting of operators which operate on relations to produce new relations. The three operators that are most commonly used in query processing are projection, selection and join.

The projection operator takes a relation and one or more attributes as arguments. The result of a projection is a new relation which contains only the attributes used as arguments to the projection. For instance, in the employee relation of Figure 2, a projection on the attribute DEPT-NUMBER would produce a relation consisting of only the values of the attribute DEPT-NUMBER, 2 and 6.

The selection operator takes a relation and a predicate as arguments and returns a relation whose tuples satisfy the predicate. Using the employee relation in Figure 2, a selection of salary > 25,000 would return a relation consisting of the tuples for Brown and Smith.

The join operator is used to combine two relations.

This operator takes as arguments two relations (A and B) and an expression. The expression describes the relationship one attribute in relation A must have to another attribute in relation B. The result of a join operation is a relation consisting of tuples formed by taking the cartesian product of the two relations and then selecting those tuples that satisfy the expression. A join on the Employee Relation and Department Relation of Figure 2 where EMP-DNO = DEPT-ID is shown in Figure 3.

Join Employee Relation and Department Relation
where EMP-DNO = DEPT-ID

DEPT ID	DEPT NAME	DEPT LOC	EMP NAME	EMP ID	EMP DNO	EMP SAL
2	R&D	OR	JONES	020	2	15000
6	ADMIN	CA	BROWN	040	6	50000
6	ADMIN	CA	SMITH	006	6	30300

Result of a Join operation
FIGURE 3

The next level of abstraction after the data level is the internal level. This level describes the data storage. Data may be stored in main memory, or on disks, tapes etc. Storage other than main memory is called <u>secondary storage</u>. Although storing data in main memory is preferred, because this makes data retrieval very fast, main memory is usually not large enough to contain all the data in a database.

At the internal level the organization of the physical data is specified. Data is organized into files. A file is a collection of records stored in a specified order on a storage device. A record is a collection of fields in a specified order, and a field is simply a string of bytes. For example, consider an employee file. This file contains employee records. Each record contains the employee name, employee id, salary, department and employee birth date. The way the records are arranged in a file is called the _file organization_. There are many different possible file organizations. Three common file organizations are:

(1) sequential,
(2) indexed and
(3) linked.

More complex file organizations can be built using these organizations as basic structures.

A sequential file is simply a file of records. Records can only be added to the end of a sequential file and records can only be retrieved by first retrieving each of the records occurring before the given record.

An indexed file contains two parts. The first part is an index which contains a record identifier and the location of the record in the file. The record identifier is usually a field in the record which is known to be unique, or it can be a precomputed value. The second part of an indexed file is the area which contains the records.

The records need not be in any particular order, although some implementations of indexed files allow the user to specify an order. To find a record in an indexed file, first the index is searched for the record identifier, if it is found, then the record location indicates where the record is located. If no identifier is found, then the record is not in the file. The index is ordered by the record identifier, so that a binary search can be used to locate the record.

Another type of file organization is a linked file. In this organization each record contains a pointer or link to the next record in the file. The pointer can point to any record in the file. A linked file and a sequential file may appear to be the same, but the method used to retrieve the next record is different. In a linked file, the next record is determined by looking at the pointer, and retrieving the record at the location indicated by the pointer. In a sequential file the next record is always the next physical record in the file.

Often a file is organized in order by the record identifier, but it is also useful to have an index on another field in the file. A secondary index, often referred to as an index file, can be created which just contains the "new" index value and the address of the record in the original file. In a secondary index there can be more than one record associated with an index

value. For instance, returning to the employee file, a secondary index using the salary as the new index value is created (salary file). This file would contain one part, the index. The index would contain one entry for each different salary in the employee file and the address of the each record, in the employee file, that contains the corresponding salary. Now to print the employee records in order by salary, the salary file can be used to access the employee records in order by salary. The first entry in the salary file is read to find the addresses of the employee records with that salary, and the addresses are used to retrieve the corresponding records from the employee file.

The internal level specifies which data each file contains, the organization of the file, and whether any indexes are available for the data.

The last level of abstraction is the physical level. This level contains the system specific implementation of the file organizations, and the primitives for record retrieval from a file. Processing at this level is also called back end processing.

One of the functions of a DBMS is query processing. A query is a question posed by a user about the data in the database. In order to answer the question, the DBMS must determine what data needs to be retrieved, retrieve the data and then process the data. Because the data is

stored in secondary storage, data retrieval is usually the most time consuming part of this task. In fact, because of the relatively slow access time to secondary storage, data retrieval is several times slower than the other parts of query processing. In order to minimze the amount of time used in query processing, query optimization strategies have been developed to minimize the amount of data which needs to be retrieved.

On a disk-based secondary memory system, data retrieval is measured in terms of the number of physical disk accesses necessary to retrieve the data. At the physical level, data is retrieved in blocks. This is the number of bytes that one physical disk access returns to main memory. A block may contain one or more contiguous records from a file. For instance, assume that ten records from the employee file fit into one block. This means that a user program that is reading records sequentially from the file will only generate a disk access every tenth record, when all the records in the block have been used. For an indexed retrieval the situation is a little bit different. If the records are stored in order by the index value, i.e. the second record in the index is located next to the first record in the index, then reading the file by the index value would require one physical disk access every tenth record. However, if the records were retrieved in another order, for each record retrieved a user program might generate a separate physical disk

access, because the records might each reside in a separate block. If a file is ordered by the index value it is called a clustered index.

In query optimization two types of strategies are used:

(1) semantic knowledge and
(2) knowledge about how the data is stored.

For instance, using the database of company XYZ again, assume a user asks for a list of all duplicate employee-ids. In this case, the semantics of the database require that the employee-ids be unique values. Using this knowledge, the optimizer recognizes that the query can be answered without referring to any data.

Suppose another user requests the names of all employees making exactly $15,000. In this case, there are two possible ways the query can be answered. One way is to read all the records in the employee file, and check each record to see if the salary is $15,000. If it is, the employee name can be printed. In this case, if the employee file contained ten thousand records, and there were 10 records per block, then one thousand physical disk accesses would be required. Another way to solve this query is to use the secondary index on the salary. The secondary index can be used to find the record addresses of the employees who earn $15,000, and then only those records need to be retrieved. In this example, because we

know something about how the data is stored, we would choose the second method over the first.

Sometimes data access follows patterns. If the same data is used in several parts of the query, this is called _locality of reference_. When the data is used in the same order in several different parts of the query, this is referred to as _sequentiality of reference_. Data retrieval can be speeded up by _read ahead buffers_. The term "read ahead buffers" means that the system does not wait for a read instruction to do another physical disk access, but goes ahead and reads the next block as long as buffers are available. When data is being referenced sequentially this is useful, because the system may have the data in memory when it is needed.

We can see that clever file organization can minimize data retrieval time. A good query optimizer must "know" something about how file organization will affect data retrieval, and must have a way to compare file organizations. This knowledge can then be used to decide which files to use to minimize access time. In this thesis we will discuss an underlying mechanism to support the query optimization process. Most DBMS's have some type of query optimizer, but the knowledge the optimizer has about the data is very restricted, and usually is written into the optimization strategy. This thesis will present a general access method architecture which can be used to support a

query optimizer so that any type of file organization can be used.

1.2. BACK-END PROCESSING

One approach to back-end processing is what we will call the "canned algorithm" approach. Systems which use this approach choose an optimization strategy and incorporate it directly into the DBMS optimization code.

Astrahan and Chamberlin [AST 76] define an optimization technique which takes advantage of the data structure. Their method is used for optimizing SEQUEL [CHA 74], a non-procedural block structured query language. The optimizer attempts to optimize the query by means of any indexes present in the database. Each tuple in a relation is identified by means of a tuple identifier. This is an extra attribute added to each relation. It supplies a unique value for each tuple in the relation.

A user query is divided into blocks, with each block containing one constraint on the data to be retrieved. The blocks are categorized depending on whether an index exists for the variable being constrained and whether the variable is independent of the other blocks. Variables which are constrained in more than one block are called: correlation variables.

The query processor performs the following steps:

(1) form the query parse tree, with each leaf node representing a constraint,

(2) mark each leaf node according to whether an index exists on the attribute,

(3) determine which indexes will be useful,

(4) retrieve tuple identifiers from indexes,

(5) move up the tree merging tuple identifier lists from lower nodes,

(6) use the final list of tuple identifiers to retrieve tuples which satisfy the query.

In step 3, a useful index is defined as an index which will minimize the amount of data to be retrieved by restricting the set of possible tuples. For example, if a restriction such as "age = 50 OR salary = 15,000" is present, an index on either one of the attributes does not limit the tuples, because a tuple that does not meet the age restriction can still meet the salary restriction.

Assume we are solving the following query with the Employee and Department relations of Figure 2. We will also assume the database contains indexes on the attribute of DEPT-LOC and on the attribute of EMP-DNO. Consider the query:

> "List the names of employees in each department in Oregon."

This query is called a multi-variable query because it uses variables from two different relations. To solve this query, first a department tuple must be accessed to find the departments in Oregon, and to retrieve the DEPT-ID. The DEPT-ID is necessary to find the employees in the department. Using the DEPT-ID the Employee relation can be scanned for employees whose DNO match the DEPT-ID. Pseudo-code for the solution is shown in Figure 4.

```
For each Department tuple
  If DEPT-LOC = OREGON
    retrieve DEPT-ID
    scan Employee relation looking for EMP-DNO = DEPT-ID
  endif
endfor
```

Query Solution
FIGURE 4

In this query, the employees' names retrieved depend on the employees' departments. In SEQUEL, the department is labeled as a correlation variable. SEQUEL handles a query of this type be resolving the inner block (in this example, the employee relation is in the inner block) once for each tuple in the outer block (the department relation is in the outer block). The SEQUEL algorithm takes the following steps:

(1) using the index on location, find the tuple identifiers for all departments in Oregon;

(2) retrieve the department tuples;

(3) For each department tuple

   (a) using employee index on DNO find all tuple identifiers with matching DNO,

   (b) access the employee records to retrieve the employee name.

The only knowledge used in this optimization process is the knowledge about which files have associated indexes. Other file types, such as linked files, are ignored, they are assumed to be of no value in optimizing queries.

Decomposition is the strategy employed for query processing in QUEL [WON 76] [STO 76]. The query is reduced to a sequence of single variable queries which are then solved. The order in which the single variable queries are solved is important in terms of the amount of data retrieved. The algorithm uses the variable with the smallest domain to determine which single variable to solve for first. (The set of valid values an attribute can be assigned is called its domain). Although in general this is a good heuristic, it does not take into account the size of any intermediate relations that may be created, nor does it consider the cost of access when choosing a variable to solve for.

In summary, the canned algorithm approach has several advantages:

(1) because the optimization knowledge is coded into the system, it runs quickly

(2) if the scheme chosen fits the data set, (i.e. queries used in the system are queries for which the optimization algorithm produces optimum solutions), the system will be very responsive to user queries.

This approach has several disadvantages:

(1) new file types cannot be used. A system of this type can only handle file organizations which have been anticipated in the optimization strategy. Other file organizations cannot be used because they are not supported at the internal level. Even if they were supported at the internal level, the optimization strategy could not include knowledge about new file types in optimizing process.

(2) If queries are used for which the optimization strategy produces poor results, the system will appear slow and unresponsive to the user. Unfortunately, in this situation the only way to improve performance is to stop asking such queries. A more drastic approach is to redesign

the entire system, so that it handles that type of query better.

If an optimizing strategy existed which always produced an optimum solution, the canned algorithm approach would have merit. Unfortunately there is no single optimizing strategy which will always produce the best way to solve a query.

Yao [YAO 79] describes four basic tasks which are used in queries. These tasks are:

(1) restriction,
(2) record access,
(3) join and
(4) projection

He divides query processing strategies into seven classes based on the order in which the tasks are performed (see Figure 5). Each class has sub-classes based on the position of the projection operation.

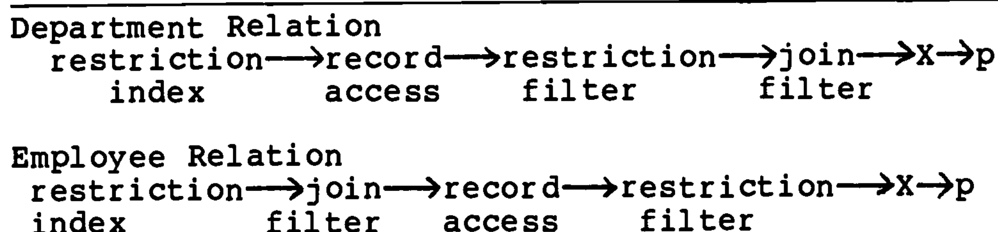
CLASS	ORDER	CLASS	ORDER
I	Restrict-Access-Join	IV	Access-Restrict-Join
II	Join-Access-Restrict	V	Access-Join-Restrict
III	Join-Restrict-Access	VI	Link-Access-Restrict
	Restrict-Join-Access	VII	Link-Restrict-Access
			Restrict-Link-Access

Yao's seven classes
FIGURE 5

Yao develops formulas for estimating the cost of query processing, and uses them to calculate costs for each

class. His results show that no class is always the optimum solution.

Using Yao's scheme, the SEQUEL algorithm for multivariable queries, discussed previously, is shown in Figure 6. The algorithm for processing the Department relation is a Class I version 1 algorithm. (Version 1 puts the projection last). The algorithm for processing the Employee relation is a Class III version 1 algorithm.



SEQUEL algorithm in Yao's scheme
FIGURE 6

Yao shows that in the special case where a clustering index exists on the restriction attribute and a non-clustering index exists on the join attribute the Class III version 1 algorithm should be applied to both relations.

Blasgen and Eswaren [ESW 77] compared four methods of evaluating a query using the operations join, restriction and projection. The methods were 1) indexes on join columns 2) sorting both relations 3) multiple passes, (i.e. one relation is stored in core and all processing except joins is done, then the join is performed on the

results) and 4) a TID (tuple-id) algorithm. In the TID algorithm all tuples are referenced by indexes. The indexes are used to produce lists of tuples which meet the restrictions in the query. The lists are sorted and then used to perform a join.

The comparison of these four query processing strategies was based on the cost of access to secondary storage. The costs depended on whether indexes existed, how selective the restriction was and whether or not the index was clustered. Formulas were developed for each evaluation method, and the authors showed that the best method to use depended on the query and the database. As with Yao's paper, we see that no single strategy will always produce the optimum query solution.

We can see from these results that the canned algorithm approach will always have serious problems which ultimately make it a poor solution for database processors. The problem centers around the fact that there is a possibly infinite number of file structures. A given file structure may be optimal for one or more types of data/query combinations. No approach to query optimization which uses a fixed set of file structures will produce better results than the result produced for a specific query in a specific database. Because the canned algorithm approach is an attempt to use a fixed set of file structures, it will only perform well in those situations

where it happens to be the optimum algorithm for the query posed by the user. A system which overcomes these problems needs to be flexible in handling file access. Ultimately what is needed is a query processing system which can support addition of new access functions without the need to rewrite the optimization strategy. In other words, a system which can support an optimzation strategy which allows arbitrary file structures will always out perform optimization schemes based on a fixed set of file structures.

Senko [SEN 69] explored the idea of basing access methods on paths to be followed in locating data. Instead of describing data in terms of files, records and fields, his method described all data in a hierarchical format. The format consisted of objects. Each object consisted of data and one or more pointers. By following the pointers, a path through the data is defined. Senko used the term <u>access path</u> to describe data retrieval. The access path described how to find data in secondary storage, that is, which pointers to follow. Several paths could be described for the same data type. Senko then described an optimzation strategy which looked at all the paths that could be used to retrieve the relevant data and choose the best path. If there are many paths, choosing the best path can be a very time consuming operation, since all combinations of all paths would need to be considered.

System R incorporates Senko's ideas [AST 75]. In System R the user specifies base tables, ie, relations without index or link structures. The base tables are stored, and access paths maintained on base tables. The access paths available are _images_ (indexes), _binary links_ (which link tuples from two different relations) and _sequential access_. As in the other schemes we have seen, only a specific set of access mechanisms can be handled. When a query is processed, the optimizer classfies each SEQUEL statement into a statement type, and then looks through the system catalog to find a set of access paths which can be used. When several paths are available, an ordered list of preferred types is used to select the best path [SEL 79].

The access methods are described in ASL (Access Specification Language) [LOR 79]. After the access paths are selected, SEQUEL statements are mapped into ASL and ASL is compiled into 370 machine code. The machine code is then executed to produce a result.

By deferring the choice of which strategy to use until the query is processed, System R allows great flexibiltiy. However, there are some drawbacks to this approach. First, the optimization algorithm allows only three file organizations, sequential, indexed, and linked. The access methods for these types are embedded in this system. Second, using pre-determined ordering to select

access paths ignores any interactions between files and intermediate file size. For instance, an indexed file is always preferred over a file without an index. However, indexed files are one-to-many and produce far more target records than index values. A file without an index may be very small, or produce values which are unique (primary keys). In this situation, the file without the index may be preferred because the intermediate file size is smaller.

Hawthorne and Stonebreaker [HAW 79] studied the effect which extended storage devices, multiple procesors and pre-fetching databanks have on the performance of database systems. They were particulary interested in what CPU functions to distribute and when and how to buffer I/O. Their research showed that different strategies need to be employed for different types of queries.

Overhead intensive queries required little time to actually get the data, but more time for processing. For example, in a database with an index on employee number, a query such as

"Find the employee name whose id is 43."

may require more time to parse the query and examine system catalogs than to fetch and manipulate the data. This type of query shows little locality of reference, other than system catalogs. For overhead intensive queries, the processing should be distributed at the terminal monitor

level and extended storage (segments allocated from main memory) used to cache system relations.

Other queries require more time to process the data than the time needed for system overhead. There are two reasons for this:

(1) the query itself is data intensive, (i.e. "list all employees") and

(2) the database structure is inappropriate for processing the query. For instance, the former example becomes data intensive if an index on the employee number did not exist.

Data intensive queries show locality of reference and sequentiality of reference. The authors suggested two approaches:

(1) distribute the processing at the data level for data intensive queries, using extended storage for relations which are accessed repeatedly during processing of aggregate functions

(2) use read ahead buffers to take advantage of sequential reading of data.

The authors identified a third type of query, the multi-relational query. This query needs information from more than one relation in the database. These queries are CPU bound. The authors suggested distributing the

processing at the data level and caching relations which are re-accessed repeatedly in extended storage, or invoke read ahead buffers. Once again the strategy to be used varies radically with the query type or the file structure.

In summary, we have seen repeatedly that no single optimization strategy will produce an optimum result for any query. Because of the infinite number of file structures, and the fact that many of these are optimal in certain precisely limited situations, optimization strategies which can only handle a fixed set of file organizations will not produce optimum strategies for arbitrary queries. An architecture which supports arbitrary access paths will always have the potential of producing better results than one whose optimization schemes are based on a fixed set of file structures.

We can see from this overview that query processing has progressed from a static process to a more dynamic, flexible process. Moving large amounts of data is a time consuming process, and the trend is to use as much information about the nature of the data as is available to minimize the amount of data retrieval. In the ensuing chapters we will define an architecture which provides a greater degree of flexibility by allowing unrestricted file organizations to be used in the optimization process.

1.3. THE DATAFLOW APPROACH

The conceptual basis of our proposed solution to this problem will be the dataflow approach to computation. The basic conceptual approach of the dataflow metaphor is to view data as a stream of values [LAN 65] between totally independent operators. Each operator can only affect data in the streams which pass through it. The operator affects a stream by adding, changing, or removing values as they pass through.

The dataflow metaphor can be used at three different levels:

(1) as a framework for computer architecture [TRE 82],

(2) as a framework for language [GOS 79] and

(3) as a framework for describing generic processing strategies.

In a dataflow architecture, a computation is conceptualized as a collection of operators which are activated by incoming data. The data is visualized as a stream flowing from operator to operator. When data for an operator is present, the operator takes the data from the input stream, processes the data and places the result, i.e. the output, into an output stream "flowing" toward the next operator.

Dataflow architectures are data oriented. There is no explicit invocation of operators. Rather each is triggered when an input value becomes available. Operators in

a dataflow architecture may be dedicated to a particular processor, or may be available to be run on one of several processors.

In contrast with a Von Neumann architecture, a dataflow architecture has several important differences. First, data is viewed as a stream [LAN 65] which is acted upon by operators. This is much different than the Von Neumann approach which views data as global to all instructions, since any location in memory is available to any instruction. In the Von Neumann architecture a modification made to data is not always apparent as the output of that instruction. In a dataflow architecture a modification is always apparent because the output value is different from the input value.

This view of data means that dataflow languages are applicative languages. All processing is done by means of operators applied to values. "Side effects" are prohibited, making programs easier to debug and design, because module interaction can be determined by looking only at the inputs and outputs [GOS 79]. All interactions must be explicit.

Second, dataflow architectures are easily implemented on parallel machines. In the Von Neumann architecture, the central processing unit is operating on a sequential set of instructions. Proper execution of each instruction may not be dependent on the execution of preceding

instructions. For example, if several variables need to be initialized to zero, these instructions could be performed concurrently, since they are independent of each other. In a Von Neumann architecture this independence of instructions is not evident from the program structure alone. In a dataflow architecture the design of the architecture makes concurrent processing easy to implement. Since an operator waits for data to be available before it begins processing, operations which do not share a common data stream are temporally independent. Furthermore this independence can be determined just by looking at the program structure.

The dataflow architecture can also be used as a data structure in query processing problems. Buneman [BUN 82] describes FQL as a language to describe queries in terms of streams of data.

The advantages of using the dataflow metaphor as a framework for query processing are as follows.

(1) Each file access can be viewed as an independent and perhaps concurrent operator. Access operations thus become transparent to system organization.

(2) Operator concurrency makes parallel processing easy to implement. File accesses which are independent of each other are easily identified and can be processed independently. More generally, it is apparent which

operators can be processed independently. A query processing strategy thus formulated is not limited to a single processor implementation.

The main disadvantage of this approach is that dataflow machines are not generally available.

In the next chapters we will examine a framework for building query processing strategies which is based on the dataflow metaphor. We shall see that the transparency of access path operators in the dataflow approach yields a framework capable of handling arbitrary access methods, and thus of superior flexibility and performance potential to those discussed in this chapter.

A DATAFLOW FRAMEWORK FOR ACCESS PATH DEFINITION

As we have argued in Chapter 1, a query optimizer requires the ability to reference access paths independently and to put access paths together to produce optimum query solutions. Because database queries have arbitrary structures, the flexibility and ultimately the performance of a query processing strategy will depend on its ability to build arbitrary structures for arbitrary access paths. In the remaining chapters we will see how query processing strategies can be defined on a virtual Data Flow Access Machine and executed on a lower level File Access Machine.

This chapter describes the design and implementation of the Data Flow Access Machine. Section 2.2 will review the origins of the dataflow paradigm, and show how it fits into the entire process of query strategy selection. Section 2.3 will define Data Flow Access Modules and Section 2.4 describes the Data Flow Access Language.

2.1. OVERVIEW

In this section we will see how the dataflow paradigm for computation described in the previous chapter can be formulated as a framework for describing access path computations which are constrained only by the ordering requirements of the access paths themselves. Within this dataflow framework, the need arises to define individual access paths in terms of a set of primitive data access

operations. The fact that data accesses are treated as primitive operations allows a database designer complete freedom in defining access methods. Independence of access paths also permits an extremely flexible use of access path algorithms in the optimizing process. Since the access paths for a file are not restricted to some predetermined set, file structure can be precisely tailored for optimum performance of expected queries. The optimizer then has the ability to plan a query without restrictions on preconceived file types or ordering of file access methods. When the user (or database administrator) creates a file, that user is also responsible for describing the file organization. Normally this is performed by supplying the access path code itself.

Describing the file organization means supplying file access methods along with an explicit indication of the purpose which these access methods serve. When data needs to be retrieved, the appropriate access method is located, and the retrieval system simply executes the file access primitives. When file organization is changed, all that is required is that the description of the file access method be changed to correspond to the new organization.

This solution requires:

(1) a dataflow language for expressing query processing strategies,

(2) a language for specifying the function performed by an individual access method,

(3) a programming language suitable for defining file access methods,

(4) a machine which can execute file access primitives.

I will call the third item the File Access Language and the fourth item the File Access Machine. The first two aspects will be discussed in this chapter and the second two in Chapter 3.

2.2. QUERY PROCESSING FRAMEWORK

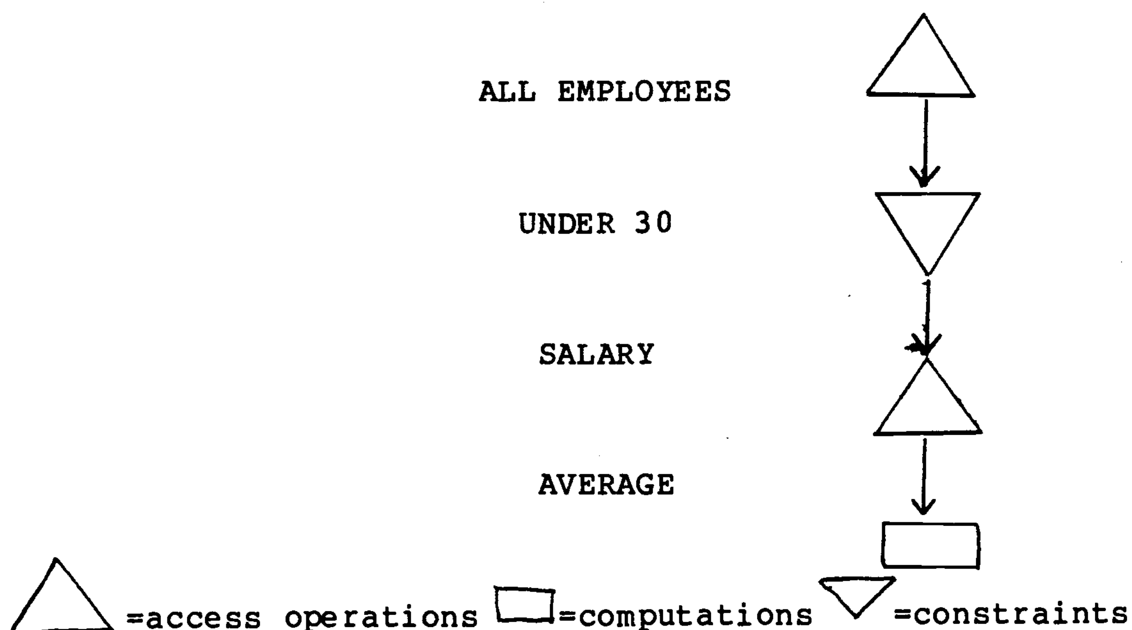
When a query is received from the user, it is conceptually composed of three items:

(1) access operations,
(2) constraints and
(3) computations.

Any plan for processing the components of a query can be described by a dataflow graph, where each node in the graph represents an operation in the query. The nodes are connected by directed arcs. Outgoing arcs represent values produced by the operation at the node. Incoming arcs represent input values for the node. Figure 7 shows a dataflow graph for the query

"Find the average salary for all emplyees under 30."

Using the decomposition described above, the dataflow graph can define any database query. If we define and provide support for a language construct called the _access module_, we can define all the code necessary to process a query as access modules. We can then define the entire query processing strategy by connecting the data streams of the access modules together.



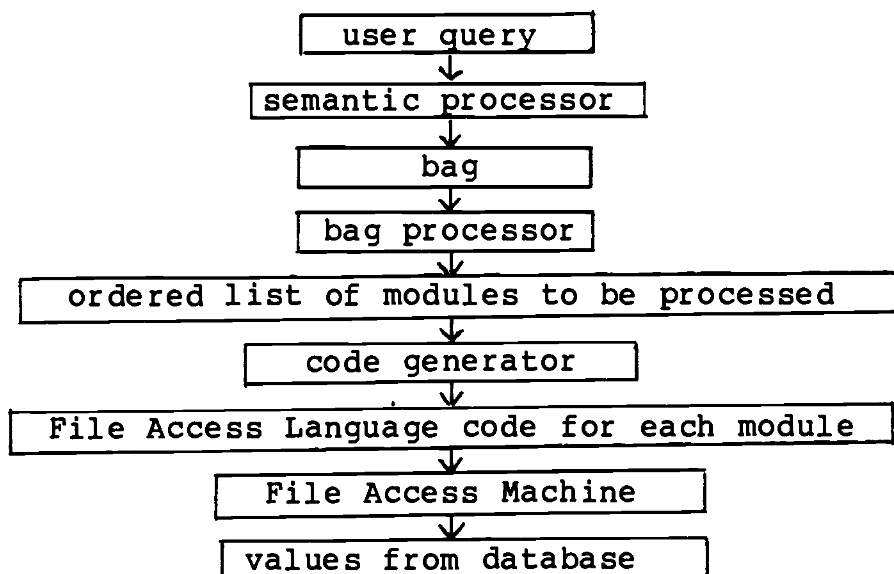
A dataflow graph for
Find the average salary for all employees under 30
FIGURE 7

This thesis was developed as part of an experimental DBMS called OSIRIS. An overview of query processing in the OSIRIS architecture is shown in Figure 8. When a user first types in a query, it is passed to the semantic

processor. The result of the semantic processing is put into a bag query and passed to the bag processor. Each unit, or bag, in the bag query contains the following information:

(1) what items to retrieve from the database,
(2) constraints and computations to perform and
(3) what items to return.

The bag processor optimizes the query by selecting and ordering the access paths which will be used in solving the query. The output of the bag processor is a dataflow network of access modules. Each module uses the output of the previous module as its input stream. The output stream of the final module contains the answer to the query.



Overview of Query Processing in OSIRIS
FIGURE 8

Each access module in the dataflow graph consists of at most one file access and any applicable constraints and computations. Each access module in the dataflow network is described by a module descriptor. The dataflow network produced by the bag processor is passed to a code expander, which uses the module descriptor to generate File Access Machine code. Finally, the File Access Machine interpreter executes this code, producing a stream of output values for the module.

2.3. DATAFLOW ACCESS MODULES

Providing a framework for query formulation requires that several practical problems be solved. For instance, if some file access method, such as an indexed retrieval, requires a value for retrieval, how is the value (or a vector of values) to be provided? If they are in a file, how will the values be retrieved from the file? Naturally, assumptions about where these values come from can not be written into the access method. Although in general one would expect temporary files to be sequential files, is is not unlikely that a database application which generates large temporary files all the time may need to deal with these files in a different manner. By keeping the access methods independent of the file structures which supply them with input values, such a change in the "feeder mechanism" can be accomplished without changing the definitions for any access methods that

require input values. In addition, the definitions could be used for any type of file in which the input values are stored.

An additional question concerns the values produced by an access method. Appropriate values are often restricted by constraints in the original query. For example, in the query of Figure 9 the constraints are "sex=male" and "hair=brown".

---

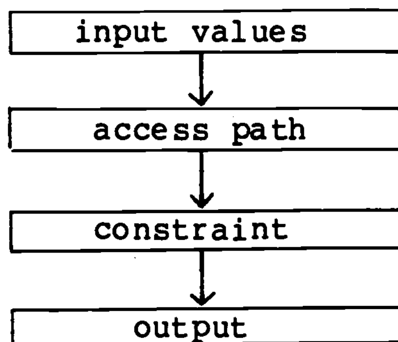
name all male employees with brown hair

Sample query
FIGURE 9

One way to deal with constraints is to first instantiate the entire set and then discard those set members that fail to meet the constraints. This strategy requires initial storage for a possibly large amount of data that may not even be used. The preferred method is to as apply the constraints to each member of the set as it is retrieved, so that storage is needed only for those items which meet the constraints. From this discussion, we see that each node in a dataflow graph based on file access modules must contain four components:

(1) handling of input values,
(2) individual access routines,
(3) enforcement of constraints and
(4) handling of output values.

Figure 10 shows the structure of an access module.



Basic Module Structure for
Data Flow Access Machine
FIGURE 10

Modules defined in our Data Flow Access Language may refer to files of three different types: input files, one output file, and a work file. Input files are received from the previous access modules and are read-only files from the viewpoint of the access modules. There may be more than one input file. The output file contains the values which will be passed to the next module. The output file is a write-only file from the viewpoint of the access module. The access module treats the input file and the output file as streams or sequential files.

The work file is an actual physical database file from which data is being retrieved via some access path. This is the only file that can be both read from and written to. Data from the input file may be used to retrieve records from the work file, and information from either file may be written out to the output file. A module will always have an output file, but the input files and work

files are optional. Unlike the input file and output file, the work file is expected to have random access capabilities. Work files provide a language interface to the permanent random access files which actually make up a database. An access module may have at most one work file.

Each access module is described by a module descriptor which contains the following information:

(1) the size of the "records" in the input stream,

(2) which values from the input stream will be used in the file access method,

(3) any constraints and computations to perform in the module,

(4) the name and size of the value the file access method will return,

(5) if an input value is used, whether it should be saved or can be discarded,

(6) whether the relationship between the input value and output values is one-to-one or one-to-many,

(7) buffer allocation for each file,

(8) a description of the work-file which the module will use and

(9) the name of the template to use for code expansion (templates are defined in Section 2.4).

The work-file description includes the name of a schema file containing the access code for the work file, information about the record size, and for each field in the record, the name, offset, and length of the field.

EXAMPLE

For example, assume we are solving the query shown in Figure 11.

"Name all male employees with blond hair."

Sample Query
FIGURE 11

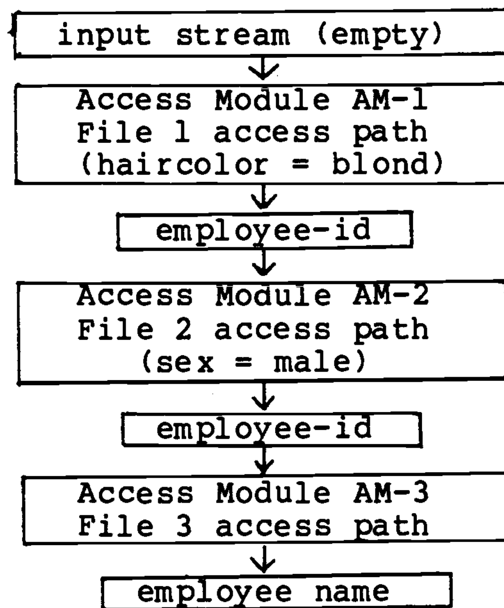
The database being used has three files. File 1 contains records which have the employee-id and haircolor. File 2 records contain employee-id, and sex. File 3 records contain employee-id and name. The schema for this sample query is shown in Figure 12.

FILE 1	[employee-id, haircolor]
FILE 2	[employee-id, sex]
FILE 3	[employee-id, name]

Sample Schema
FIGURE 12

The query optimizer divides the query into three modules. The first module will retrieve the employee-id of all employees and produce the employee-ids of those employees with blond hair. The second module will receive the employee-ids from module 1 and retrieve the sex of these employees. The records which meet the criteria sex=male will be selected, that is, the employee-ids will be placed in the output file for processing by module 3. The third

module will use the employee-ids from module 2 to retrieve the names. The dataflow graph for this query is shown in Figure 13.



Data Flow Access Graph for sample query
FIGURE 13

The access module descriptor for access module AM-1 is shown in Figure 14. The access path information contains information about the work file. In this example, the code which describes the access path is contained in two files named EMP-HAIR-CMD and EMP-HAIR-VAR. The work file will generate at most one record for each input value. The information about the record length and fields is also provided. This module has no input field. The output field is emp-id which is located in the work file.

This access module contains one operation to perform,

ACCESS PATH INFORMATION

access-path-file-name: EMP-HAIR-CMD

variable-file-name: EMP-HAIR-VAR

file-type: generates one record per input value

record length: 10 bytes

fields in file:

emp-id: integer, 5 bytes long, offset 0

emp-hair: character, 5 bytes long, offset 5

INPUT-FIELD FOR MODULE: none

OUTPUT-FIELD FOR MODULE: emp-id, location is work-file

OPERATIONS TO PERFORM:

equal bytes

operand 1 is emp-hair

operand 2 is a constant named MODULE-CONSTANT1

BUFFERS TO ALLOCATE

output file: 1

work file: 1

TEMPLATE: A

Module Descriptor for Module AM-1
FIGURE 14

an equality test. The fields which should be compared are emp-hair and a constant named module-constant-1.

The access module descriptor also specifies how much buffer space should be allocated to each file. In the example, the output file and the work file should each

receive one buffer. Finally, the module descriptor contains the name of the template which should be used when the module is executed. In this example Template A will be used. (Module descriptors for access modules AM-2 and AM-3 are shown in Appendix D).

2.4. DATA FLOW ACCESS LANGUAGE

The implementation of dataflow networks is achieved by using a Data Flow Access Language. The Data Flow Access Language is an extension of the File Access Language to be discussed in Chapter 3. It allows a high level integration of input file access, work file access and output file access. The Data Flow Access Language is composed of the operators of the File Access Language and a set of meta-commands. The meta-commands indicate a generic action whose details depend on the structure of the input stream, output stream, or file access method. Since these details are not known until a query is processed, the appropriate access commands cannot be included in the template. Instead, their interpretation is deferred until a module is actually executed. Figure 15 shows a list of the meta-commands.

Using the Data Flow Access Language, access module algorithms can be constructed which are independent of the specifics of input and output files. For example, the pseudo-code for the algorithm to process the access module AM-1 in the previous example is shown in Figure 16. Not

---

1) DOP -- do-operations
2) WRO -- write-output
3) WLB -- write last buffer
4) DFO -- do final operations
5) INI -- file initialization
6) CLO -- close files
7) GNI -- get next input record
8) GNIF -- get next input field
9) GNW -- get next work record
10) GNWF -- get next work field.

Data Flow Access Language Meta-Commands
FIGURE 15

```
allocate the buffers
open the files
get a record from the work file
while there is a record
    apply constraints and do computations
    if constraints are met write the output record
    get the next record from the file
end while
do any final processing
close the files
```

Algorithm for processing an access module
FIGURE 16

---

only will this algorithm process the access module AM-1, it will also process any access module that uses no input stream. Algorithms which describe module processing independently of the file specifics are called templates. These templates are written in Data Flow Access Language. Figure 17 shows the template for the algorithm shown in Figure 16. The '+' sign is used to mark meta-commands which will be expanded when the template is used. Other

instructions are from the lower level File Access Machine (see Chapter 3). This template is used to process the first module for our example query. Actual Data Flow Access Language Code for the templates used in the processing of the other modules in the example query is shown in Appendix C.

```
+    INI            allocate buffers and open files
+    GNW            get the first record from the work file
     WHILE M1 0     while there are records in work file
     EQ C_EOF 0
+       DOP                   do module operations
        EQ W_FLAG TRUE         check if write flag is set
        IF M2 M2               if write flag is set
+         WRO                   write output record
        M2                    end if
+       GNW                    get next work record
     M1                    end while
+    DFO                   do final operations
+    WLB                   write last buffer
+    CLO
     RET 0 0
```

Template A
FIGURE 17

The actions indicated by Template A involve reading a record from the work file and then performing whatever operations are specified in the module descriptor. (The DOP meta-command is expanded into the appropriate operations by the Dataflow Code Expander). If the WR-FLAG has been set as a result of these operations, the output record is moved to the output file. If the WR-FLAG is not set, then the corresponding value for the output record is not moved to the output file. When the end of the work

file is reached, final operations are performed, the last buffer is written to the output file, and the files are closed.

THE FILE ACCESS MACHINE

In Chapter 2, it was shown that queries can be described using dataflow graphs. Access modules were introduced as a framework for describing a dataflow graph. In Chapters 3 and 4 we will discuss how the Access Modules are implemented.

The Access Modules described in Chapter 2 are intended for implementation on a processor called the File Access Machine. This chapter defines the character of the File Access Machine and its associated File Access Language. Chapter 4 describes the process by which access module definitions from the virtual Data Flow Access Machine are translated into File Access Machine code.

3.1. THE FILE ACCESS MACHINE

The File Access Machine was developed as a processor to implement dataflow graphs. It is an atomic Von Neumann processor. It is intended as a virtual machine specification for a file processing machine.

The File Access Machine is composed of an interpreter and a set of operators which the interpreter recognizes (see Figure 18 for a list of the operators). The machine has a Von Neumann architecture, and provides underlying support for implementation of the dataflow modules discussed earlier. It executes the instructions in a sequential fashion, using a pointer to keep track of the next

operator to be executed. The user program which the File Access Machine interprets is stored in a program area. The variables and constants used by the program are stored in a separate area, called the symbol table. The File Access Machine operators have two operands. The operators act on global registers. The format of the operators was chosen to simplify code generation and interpretation.

Arithmetic operators: ADD, SUB, MUL, DIV, MOD
Control operators: IF, WHILE, PUSH, POP
Logical operators: AND, OR
Data Movement: MOV, MOVB
Conditional operators: EQ, EQB, GT, GTB, LT, LTB
I/O operators: OPEN, CLOSE, GET, COPY, WRITE
SKIP, ALLOC, CLEAR

Operators for File Access Machine
FIGURE 18

3.2. RUN TIME ENVIRONMENT

A File Access Language program consists of two parts, a symbol table and executable File Access Langauge code. The symbol table contains the definitions for the user declared variables and the interpreter registers. A symbol definition has the form:

[name, type, value]

The name identifies the variable. The type must be one of the following: integer, character, integer pointer or character pointer. The value is the initial value for the

variable. Figure 19 shows a sample symbol table.

variable name	type	value
w-flag	1	0
w-buf	2	0

type 1 = integer   type 2 = integer pointer

Symbol Table
FIGURE 19

The File Access Machine has two kinds of registers: public and private. Public registers are registers which are set by File Access Machine commands, and can be tested and changed by the File Access Language. Private registers are used by the File Access Machine, but are invisible to the user.

The private registers are shown in Figure 20.

```
op-ptr --- instruction pointer
s-ptr --- internal stack
c-ptr(i) --- code pointers
d-ptr(i) --- data pointers
b-ptr(i) --- buffer manipulators
```

Private Registers for File Access Machine
FIGURE 20

The _instruction pointer_ is used to point to the next File Access Machine instruction to be executed. The

internal stack is used to save the value of the instruction pointers stored by the PUSH instruction, and to restore the value of the instruction pointer in executing the POP instruction. _Code pointers_ point to the beginning and end of the area where the user program is stored. This is a contiguous area in memory. _Data pointers_ point to the beginning and end of the symbol table. _Buffer manipulators_ are indexes into the buffer table. The buffer table is used to keep track of the buffer space used by the I/O operators.

The public registers used by the File Access Machine are listed in Figure 21. A list of the operators and the public registers they set is included in Appendix B.

The register named _a-result_ is the arithmetic register. All arithmetic operators (ADD, SUB, DIV, MUL, MOD) place the result of the arithmetic operation in a_result.

The register _a-length_ is used by the string comparison operators (GTB, LTB, EQB). These operators expect the length of the strings to be in the register a-length when the operation is invoked.

When an error condition occurs, the register named _c-error_ is set. All I/O operators except CLEAR can set the c-error register.

The test register is _c-code_. This register is set by the comparison commands (EQ, EQB, GT, GTB, LT, LTB) and

the logical commands (AND, OR). IF and WHILE commands check the value in c-code to determine their execution path. The logical commands also test the value in the c-code register to determine what the final value in the register should be. Intuitively, a 1 value in the c-code register indicates Boolean _true_ and a 0 value indicates Boolean _false_.

The I/O commands use two public registers, c-fd and c-eof. The register _c-fd_ contains the current file descriptor. The CLOSE, COPY, GET and WRITE commands use the value in c-fd to specify the file which should be used. The OPEN command returns the file descriptor in the c-fd register. The register _c-eof_ is set automatically by the File Access Machine when the end of a file is reached as the result of using the COPY or GET command.

FILE ACCESS MACHINE REGISTERS
PUBLIC REGISTERS

a-result	c-code
a-length	c-eof
c-error	c-fd

Public Registers
FIGURE 21

The File Access Machine interprets operations defined by the File Access Language, which we will see shortly. The interpreter executes an operation by fetching the

value of the command arguments from the symbol table and then applying the code which defines the operator. Each File Access operator is responsible for setting a new instruction pointer value upon completion.

The user program consists of a list of variable declarations and a sequence of File Access Language commands to be executed. The interpreter loads the variable declarations into a symbol table, and the commands into a command-table. Execution begins at the beginning of the command table and continues until a "RETURN" command is encountered or the operator pointer is pointing outside the bounds of the command table. (see Figures 40 and 41 for a sample program)

The machine has two modes: _load mode_ and _interpret mode_. In the _load mode_ the variables declared by the user are loaded into memory. The variable name is placed in memory and space is reserved to store the value of the variable. The variable is marked to indicate its type. After the variables are loaded the commands are loaded into memory, and the beginning and ending addresses of the command memory segment are saved.

When the loading phase is complete, the interpreter is ready to begin. In the interpretive mode, the commands are executed. An instruction pointer indicates the next command to be executed. When a command is executed, the following sequence of events occurs:

(1) the argument type is checked to make sure it is valid for the command invoked,

(2) the values of the arguments are fetched and

(3) the code implementing the command is executed.

The interpreter continues executing commands until a "RETURN" command is reached or until the instruction pointer contains an address that is outside the code segment. The File Access Machine operation is summarized in Figure 22.

```
load variables
load instructions
set operator pointer to point to first instruction
while operator pointer points to valid instruction
     fetch argument values
     execute operator
end while
```

Outline of File Access Machine
FIGURE 22

3.3. THE FILE ACCESS LANGUAGE

The File Access Language defines a set of primitive instructions which can be used to define arbitrary file access methods. The current emulator for the File Access Machine uses a PDP-11/44. Primitive instructions are defined as routines in the language "C". The current emulator was designed as a development vehicle to provide maximum flexibility. Other feasible implementations could

include macro-style expansion of operations, emulation at a microcode level, or even special processor design.

Many of the primitives in the access machine language are similiar to those found on most processors. What distinguishes this language from other programming languages however, is a special set of file I/O commands to handle file buffer manipulations. This group includes buffer allocation and manipulation, opening and closing files, and reading and writing files.

The File Access Language uses variables and constants for arguments to the access commands. The variable may have one of the following types: _integer_, _character string_, _pointer to an integer_ or _pointer to a character string_. A File Access Language program consists of variable definitions and the File Access Language commands. Figure 23 shows a partial File Access Language program. Lines 1 thru 3 are variable definitions. Line 13 uses a public register, which does not need to be defined. Line 15 uses a variable defined by the user. The executable code begins at line 10.

A brief description of the primitive operations follows. For more detailed definitions see Appendix A.

```
 1. w-flag     1,0
 2. w-buf      3,0
 3. o-buf      3,0
 |
 |
 |
10. psh     INI,0          call initialization routine
11. psh     GNW,0          get a record from the work file
12. whl     ml,0           while not end of file
13.   eq     c-eof,0
14.   psh    DOP,0            call do operation routine
15.   eq     w-flag,true
```

Partial File Access Language Program
FIGURE 23

3.4. FILE ACCESS LANGUAGE OPERATIONS

3.4.1. CONTROL STRUCTURES

The control structures permitted by the File Access Language are:

(1) IF THEN ELSE ENDIF
(2) WHILE
(3) PUSH
(4) POP

IF and WHILE depend on the value of the condition register, _c-code_, to determine the sequence of commands to execute. IF and WHILE both use the condition register _c-code_ to signal the result of evaluating an expression. A value of true causes the IF statement to execute until it comes to the ELSE statement. Then control is passed the the next instruction after the ENDIF. A value of false causes control to be passed to the statement following

ELSE. The WHILE loop continues execution until the value in c-code is false. PUSH saves the current address in the instruction pointer on the internal stack, and changes the instruction pointer to the value that is the argument of PUSH. POP places the value at the top of the internal stack in the instruction pointer. The PUSH and POP commands are included to support procedure calls within a File Access Language module.

3.4.2. ARITHMETIC STATEMENTS

The arithmetic primitives are addition, subtraction, division, modulo and multiplication. Only integer operands are implemented in this version. The arithmetic commands all place the result in the register a_result.

3.4.3. TEST OPERATORS

The test operators are equal, greater than, less than. There are two sets of test operators, one for numeric tests, (signed) and the other set for string tests (unsigned). The integer operators are: EQ, GT, LT. The string operators are: EQB, GTB, LTB. All string operators expect the register a-length to contain the length, in bytes, of the strings. Both integer and string test operators expect both arguments to have the same type. If the test indicated is true, the register c-code is set to true, otherwise, c-code is set to false.

3.4.4. FILE PRIMITIVES

The file primitives are: ALLOCATE, CLEAR, CLOSE, COPY, GET, OPEN, SKIP, and WRITE.

ALLOCATE allocates buffer space for COPY, GET, and WRITE. When buffer space is allocated, the number of bytes requested is reserved for the buffer. Next, a private buffer record is created for the buffer. The buffer record contains the beginning address of the buffer, and a buffer pointer which marks the current point of operation within the buffer. The buffer pointer is set to zero. A request for buffer space returns the address of the buffer record. The buffer pointer can be set using the SKIP command. After execution of the COPY and GET commands, the buffer pointer is set back to zero. CLEAR returns buffer space to the system. The programmer must explicitly return buffers to the system. This allows the program to write one buffer to several different files.

GET and COPY both retrieve information from the disk. The register c-fd determines which file is read from. COPY will read the next sequential block in the file into the buffer and automatically increment the file pointer. GET will perform a read after random positioning in the file. Both commands reset the corresponding buffer pointer (b-ptr(i)) to the beginning of the buffer. If the block is non-existent, or the end of file is reached the register c-eof is set.

The OPEN command sets the register c-fd, and opens the file if it is not already open. CLOSE closes the file.

WRITE writes a buffer to a file. The register c-fd determines which file is written to.

3.4.5. DATA MOVEMENT

A MOVE command is included to move data to other locations. There are two versions, one for integer moves, MOVE, and the other, MOVEB for string moves. MOVEB expects the length of the string to be in the register a-length.

DATAFLOW ACCESS LANGUAGE IMPLEMENTATION

In this chapter we will describe how an access module is expanded into File Access Language instructions which can be executed on the File Access Machine. As we have seen in Chapter 2, the query solution is described by access modules. Once the solution to a query has been described, the next step is to expand the access modules into File Access Language. For this the following items are needed:

(1) the appropriate access path file,
(2) the access module descriptor,
(3) a template and
(4) the code expander.

The access path file is the mechanism by which file access operations are treated as an abstract data type. The access path file contains definitions of file I/O operations which are used for the work file. The following operations are defined:

(1) OPEN -- opens the file;
(2) CLOSE -- closes the file;
(3) DELETE(rec) -- deletes the record from the file;
(4) ADD(rec) -- adds the record to the file and
(5) RETRIEVE(type) -- uses the indicated retrieval method to retrieve a record from the file.

As was mentioned earlier, the database designer provides these definitions using File Access Language. The code in the access path file is used when the access module descriptor is expanded.

As we showed in Chapter 2, a query solution is described using access modules. Each access module is

described by a module descriptor. The module descriptor contains information about

(1) the work file,
(2) operations to perform in the module,
(3) input and output fields for the module.

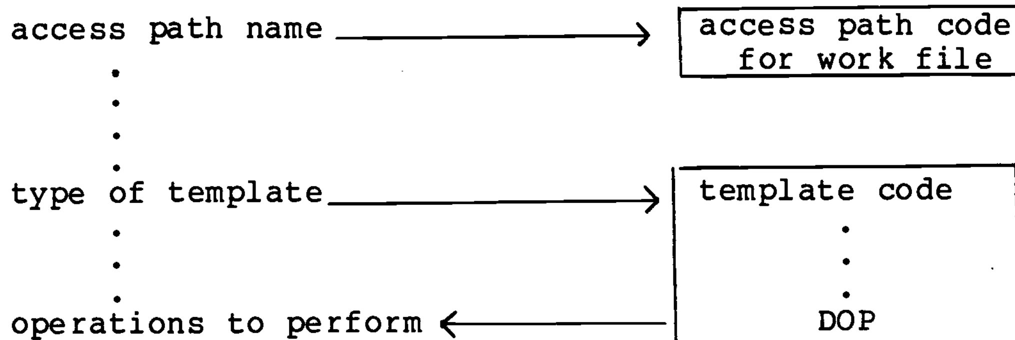
In other words, the module descriptor contains the specific I/O file and format information which is needed to process the module.

The module descriptor also contains the type of template to use when the module is processed. As we saw in Chapter 2, each template describes an algorithm to use in processing modules. However, the description contains meta-commands to indicate generic actions whose details depend on the input stream, output stream, or file access method. The actual code for an access module is assembled by replacing these meta-commands with specific file access operations (from the access path file) and specific computation operations and I/O formats (from the access module descriptor).

There are three different templates. They are named Template A, Template B and Template C. The choice of which template to use depends on whether or not an input value is needed for the work file record retrieval and on the number of records each input value generates. Template A expects no input values. An example of its use would be for sequential retrieval. Template B expects an input value and also expects each input value to produce

no more than one record. An example of its use would be an indexed file retrieval when the key is a unique value. Template C also expects an input value. However, in this case record retrieval can produce more than one record for each input value. An example of its use would be an indexed file retrieval when the key is not a unique value. Appendix C contains the template definitions. The relationship between the module descriptor, the template and the work file is shown in Figure 24.

In this chapter we will describe how the module descriptor is used to expand the template into File Access Language which can be executed on the File Access Machine.



Use of Module Descriptor
Figure 24

4.1. CODE EXPANDER

The Dataflow Code Expander prepares an access module for execution by converting Data Flow Access Language code

into File Access Language code. The Dataflow Code Expander uses the module descriptor as input. The module descriptor provides the specifics for the query, the input and output fields for the module, and the operations that need to be performed. The code expander uses templates to produce the code. These templates embed individual file access operations in support code which manages the retrieval and transmission of data values. The Data Flow Access Language commands are expanded into File Access Machine code which describes how the operations are to be performed.

As we have stated previously, a dataflow graph of access modules is used to describe the query processing strategy. When an access module is expanded to File Access Language code, its incoming data stream becomes the input file, and its outgoing data stream becomes the output file. The input file and output file for an access module are temporary files used for the module's processing.

The code expander produces a program in File Access Language code to process each access module. The File Access Language program contains a symbol table and executable instructions. This program refers to a set of pre-defined variables. These variables are defined in the symbol table. The variables are shown in Figure 25. The operand pointers are used for arithmetic and test opera-

```
operand pointers: op1, op2, cop1, cop2
output record size: or-sz
write flag: wr-flag
buffer pointers: obuf, ibuf, wbuf
current record pointers: w-ptr, i_ptr, o_ptr
field pointer: f-ptr
```

Data Flow Access Machine Variables
FIGURE 25

tions. Variables op1 and op2 are used to hold integer operands for arithmetic operations. Variables cop1 and cop2 are similiarly used for string operands. The write flag is used to indicate if the current record should be written to the output file. The buffer pointers are used to point to the buffers for the output file, the input file and the work file. Each file has a current record pointer, which points to the beginning of the last record accessed in the file.

The code expander also creates a description of the output file. This description is not used by the File Access Machine. Instead the code generator uses the description of an output file to generate code in the next module which will accept it as the input file. A file description contains the name, offset, size and type for each field in the file. This information is used by the code expander in generating code. For instance, the module descriptor may specify that the "employee-id" is the input field. Using the input file description, the

code expander knows how to calculate the address of that field from the beginning of the input record.

4.2. DATAFLOW META-LANGUAGE

Meta-commands are those commands available in the Data Flow Access Language which are not recognized by the File Access Machine. As we saw in Chapter 2, the meta-commands indicate a generic action whose details depend on the structure of the input stream, output stream or file access method. Since these details are not known until the access code is actually executed, the appropriate access commands cannot be generated when the access code is written. Instead, their interpretation is deferred until an access module is actually executed. These commands build access machine code tailored to the module which is being processed by the code generator. In general the meta-commands mark places in the template where information specific to the query must be known, such as what is the offset of the input value or what constraints are to be applied, etc. Figure 26 shows a template definition in Data Flow Access Language. This template makes use of meta-commands to perform a sequential sweep through a file and output all records which meet the criteria specified in the access module descriptor. The Meta-commands are marked with a '+'. A list of the meta-commands is shown in Figure 27.

```
+    INI              allocate buffers and open files
+    GNW              get the first record from the work file
     WHILE M1 0       while there are records in work file
     EQ C-EOF 0
+       DOP                      do module operations
        EQ W-FLAG TRUE            check if write flag is set
        IF M2 M2                 if flag is set
+          WRO                     write output record
        M2                       end if
+       GNW                       get next work record
     M1                       end while
+    DFO                      do final operations
+    WLB                      write last buffer
+    CLO
     RET 0 0
```

Template A
FIGURE 26

1) DOP -- do-operations
2) WRO -- write-output
3) WLB -- write last buffer
4) DFO -- do final operations
5) INI -- file initialization
6) CLO -- close files
7) GNI -- get next input record
8) GNIF -- get next input field
9) GNW -- get next work record
10) GNWF -- get next work field.

Data Flow Access Language Meta-Commands
FIGURE 27

The code expander uses a subroutine table to keep track of which meta-commands have been used. The first time each meta-command is encountered in the template its name is added to the subroutine table.

An initialization stack is used to store buffer allo-

cation commands. The meta-commands GNI (Get Next Input Record), GNW (Get Next Work Record) and WRO (WRite Output record) generate access code to allocate buffers the first time they are used. This code is added to the initialization stack which is eventually added to the expanded module as part of the initialization subroutine.

Meta-commands are implemented as subroutines in the File Access Language. This means when a Meta-command is encountered in the template it is replaced by a subroutine call. The first time each meta-command is encountered any variables used by the meta-command are added to the symbol table and the meta-command name is added to the subroutine table. When the template processing is complete the necessary meta-command subroutines are added to the access module's code segment. The process of producing code which executes a particular query can then be thought of as chosing a _query independent_ template and linking it to _query specific_ operations defined in the module descriptor and access path files.

4.2.1. DO OPERATIONS

This command specifies that any computations specified in the module should be included in the expanded code file at this point. Figure 26 has a DOP meta-command at line 5. For each computation specified in the module descriptor, the expander does the following:

(1) the location of the operands for the operation is found by looking in the input file description;

(2) the File Access code is generated to move the operand to the appropriate variable.

For example, assume that one operation requested in the access module descriptor is EQB, string equality. This operator uses the variables COP1 and COP2 as operands. The specific operands are found by looking in the module descriptor under Operations to perform (see Figure 14) for the module being executed. In this case, let's assume the first operand is the "employee-name" which is offset 20 bytes from the beginning of the input record. This is being compared with the field "mother's-name" which is offset 60 bytes from the beginning of the work record. Both fields have the same length, 31 bytes. The expander expects the variable i-ptr to point to the beginning of the current input record, and the variable w-ptr to point to the beginning of the current work record. The code generated is shown in Figure 28. The DOP command is not used in writing access path routines. It is used only in templates, to cause a link to the code for the query specific operations requested by the access module descriptor. DOP is used only once in a template.

```
compute address        add i-ptr 20
of 1st operand         mov a-result f-ptr
move the field         mov 31 a-length
to COP1                movb f-ptr COP1
compute address        add w-ptr 60
of 2nd operand         mov a-result f-ptr
move field             mov 31 a-length
to COP2                movb f-ptr COP2
test for equality      eqb COP1 COP2
```

Generated Code --- DOP
FIGURE 28

4.2.2. WRITE OUTPUT

This meta-command generates access machine code which writes data to the output file. The output file corresponds to the data stream produced by the access module. This is a temporary file which is used for access module processing. The output file is a file used by the access module to communicate its results to the next. The data written to the output file is organized into records. Each record contains the values needed to produce one "answer" to the query, keeping in mind that most queries produce a sequence of several answers. For instance, the query

"Find the names and addresses of all employees over 30"

may produce 264 pairs of names and addresses. Each one of the pairs would be considered one answer.

This meta-command also generates a description for each field added to the output file. The field

description contains field-name, length, type, and offset from the beginning of the output record. The collection of all the descriptions is called the <u>output description record</u>. The output description record generated by the code generator for one module is used by the code generator to find information about the input file when generating code for the next module. Description records are used only by the code generator and are not needed at run time because the correct offset and type information have been assembled into the access machine code.

The WRITE OUTPUT meta-command generates access commands to move information to the output file. The following actions are performed:

(1) calculate the size of the output record;
(2) generate code to check the output buffer;
(3) generate code to write the output record.

The size of the output record is determined by adding the size of the record in the input file to any output fields mentioned in the access module. If the module has an input field, it is checked to see if it should be discarded. An input field can be marked for discarding if it is to be used in the current module only. If the input field is marked discarded, then its size is subtracted from the output record size.

Next, code is generated which checks the buffer of the output file to make sure there is enough room in the

buffer for the output record. This code is labelled CHO. This is done by attempting to increment the output buffer pointer by the size of the output record. If this does not produce an error, then the original buffer pointer value is restored. Otherwise, the contents of the output buffer is written to the output file, and the current record pointer for the output file (ob-ptr) is set to the beginning of the output buffer. The code generated is shown in Figure 29.

```
CHO mov o-buf ob-ptr          save value of o-buf
    skp o-buf or-size         skip pointer
    eq c-error true           if c_error true,
    if W1 W2                 then not enough room in buffer
WLB opn output write          open output file for writing
    add ocnt 1                increment record pointer by 1
    mov a-result ocnt
    W2
    wrt ocnt obuf             write output buffer to file
    mov o-buf ob-ptr          reset ob-ptr
    W1
    mov ob-ptr o-buf          restore o-buf
    pop 0 0
```

Code to check output buffer
FIGURE 29

Now the output buffer is ready to receive the output record, so code is generated to move all the fields in the input file to the output buffer. Next, the module descriptor is checked to see which fields in the work file need to be moved to the output file. Code is generated to move the appropriate fields to the output file.

The code generated to write the output record is divided into three subroutines, WOB, MIF and MWB. A subroutine called WOB (Write Output Buffer) moves code to the output buffer. This subroutine expects the variable ti-ptr to contain the location of the data to be moved to the output buffer, and the variable a-length to contain the number of bytes to be moved to the output buffer. If the fields to be moved to the output buffer are contiguous, then only one call is needed for each file which contains data to be moved to the output file.

The subroutine MWB (Move Work Buffer) moves the field from the work file to the output file. In the example in Figure 30, the field to be moved has an offset of 0 from the beginning of the record, and is 5 bytes long.

The subroutine MIF (Move Input Fields) handles the fields from the input file. In this example, the field has an offset of 5, so the address is calculated in line 10. The example in Figure 30 shows code generated by the WRO command. The output record contains one field from the input file, and one field from the work file.

4.2.3. GET NEXT INPUT RECORD

This meta-command generates code to retrieve the next record in the input file. The previous module has produced a file description for the input file, so the record size is known. The code generated simply moves the input

```
    psh CHO 0          check output buffer for room
    psh MWB 0          move fields from work buffer
    psh MIF 0          move fields from input file
    pop 0 0            return
MWB mov wbuf ti-ptr    set ti_ptr
    mov a-length 5     set a_length
    psh WOB 0          call write to output buffer
    pop 0 0            return
MIF add iptr 5         calculate field address
    mov a-result ti-ptr        set ti-ptr
    mov 20 a-length    set a-length
    psh WOB 0          call write to output buffer
    pop 0 0            return
WOB mvb ti-ptr obuf       move bytes to output buffer
    skp obuf a-length     adjust output buffer pointer
    pop 0 0               return
```

File Access Machine code generated by WRO
FIGURE 30

record pointer through the input buffer. Each time Get Next Input Record is called, the current pointer for the input file (ib-ptr) is incremented by the size of the input record. If the pointer exceeds the buffer size, a disk read is generated and the pointer is set to the beginning of the buffer. The code generated is shown in Figure 31.

4.2.4. GET INPUT FIELD

This meta-command generates code to find the input field and set the input-field-pointer (f-ptr) to point to the input field. The input field for a module is the value used by the access path to retrieve values. The input field is always part of the input file. The module

```
mov ibuf ib-ptr      save ibuf value
skp ibuf [size of input record]
eq  c-error TRUE
if  M1 M2            check if buffer exhausted
cpy ibuf 0           it is, so do a read
mov ibuf ib-ptr      reset ib-ptr
M1
mov ib-ptr ibuf      it's not, so restore ibuf
M2
pop 0 0              in either case, return
```

Code generated by Get Next Input Record
FIGURE 31

descriptor contains the name of the input field. The location of the input field is found by locating the offset of the indicated field in the input file descriptor. The offset is added to the current record pointer and the result is stored in the input field pointer. If the input field is a constant value, a new variable is created whose value is the constant. The input field pointer is set to the address of the variable. The code generated is shown in Figure 32. The input field is effectively "retrieved" by setting the _i-ptr_ to the beginning of the field. Comparisons, computations or copy operations can proceed from this point.

```
add    i-ptr [offset of field]
mov    c-result f-ptr
```

Code generated by Get Input Field
FIGURE 32

4.2.5. GET NEXT WORK RECORD

The work file is the physical database file from which information is being retrieved. The module descriptor contains the name of the access path file that contains the actual code for work file operations. The access path file contains definitions for the following operations: opening and closing the file, record retrieval, field retreval, record deletion, modification and addition. The file is divided into two parts. The first part contains the location of the operation definitions and the second part contains the code for the definitions. The GNW meta-command first finds the location of the record retrieval code in the access path file and then copies the code to the expanded code file.

4.2.6. GET WORK FIELD

The work field is the value that the access method returns. The code for this operation is also found in the access path file, so the meta-command only needs to move the code to the expanded code file.

4.2.7. FILE INITIALIZATION

This meta-command produces access machine code to allocate buffers for the files used in the module and to open the files used in the access module.

For input and output files, a single "OPEN" file

access command is generated. The work file is treated differently. The access path file describes the code to be used when the file is opened, so this code is retrieved from the access path file and inserted into the expanded code file.

This command also handles buffer allocation. The module descriptor indicates the number of buffers to use for each file. Code is generated to allocate the buffers.

When this command is executed, an initialization stack is created. No commands are generated at this time. When a file I/O meta-command is used (such as WRO, GNW, or GNI) a check is made to see if this is the first time the file has been referenced. If it is, code to allocate buffers for the file, and code to open the file is generated and placed on the initialization stack. (If this is the work file, the code is retrieved from the access path file). When module processing is complete, the code on the stack is added to the expanded code file.

4.2.8. CLOSE

This meta-command closes the files. For input and output files, a single "CLOSE" file access command is generated. For work files, closing code is retrieved from the access path file. The code is then placed in the expanded code file.

4.2.9. DO FINAL OPERATIONS

Many operations performed during query processing require a final wrapup process to be executed after the main task (which usually occurs in a a read loop) has been performed. Computations like AVERAGE are examples of this. When such operations are requested via the Do Operation meta-command, the operation name is saved on a special Final Operations stack. When the Do Final Operations meta-command is executed, it checks the stack. For each operation name on the stack, the appropriate code is added to complete the processing for the operation.

For example, if the operation "average" was specified in the module descriptor, the DOP meta-command would generate code to count the number of items to be averaged, and also sum the appropriate fields (see Figure 34). These actions need to be done in the read loop. DOP would also put the operation name "AVERAGE" on the Do Final Operations stack, and include the name of the variables where the count and total are stored. Figure 33 shows the contents of the Final Operations stack at this point.

AVERAGE total-age cnt

Final Operations Stack contents
FIGURE 33

When DFO is executed, it finds the average operation and

variables on the DFO stack, and completes the processing. It adds the code to finish computing the average, and puts the answer in the field indicated by the module descriptor. Figure 34 shows code generated by DFO.

```
DOP .
     .
     .
     add cntl 1                   count items
     mov a-result cntl
     add i-ptr total-age          sum age field
     mov a-result total-age
     .
     .
     pop 0 0
     .
     .
     .
 DFO div total-age cnt             compute average
     mov a-result avg-age        put answer in avg-age
     pop 0 0
```

Code generated by Do Final Operations
FIGURE 34

4.3. ASSEMBLING THE CODE

The Code Expander accepts an initial access module descriptor and builds a file containing access machine code to implement the requested access module. This file is the *expanded code file*. The module descriptor contains the name of the template to expand. This template is "read" and each access machine command is added to the expanded code file. When a meta-command is encountered in the template, it is treated as follows. The first time

each meta-command occurs in the template, it is expanded as a subroutine and entered in a subroutine table. For subsequent occurrences, a PUSH command with the appropriate label is added to the expanded code file. When the entire template has been processed, the code generator checks the subroutine table and adds access commands for each meta-command in the table. The code is added to the expanded code file with a POP command at the end to return processing to the main program.

4.4. THE CODE GENERATOR IN ACTION

In this section we will see how the module descriptor shown in Figure 35 is expanded by the code generator into File Access Machine code. In Chapter 2 we saw the dataflow graph produced for the query,

"Name all male employees with blond hair."

The module we will be discussing in this section is the first module of the dataflow graph. Figure 35 shows the Access module descriptor for the first module. This module will retrieve the employee-ids of all employees, and produce the employee-ids of those employees with blond hair.

This module requires a template which expects no records in the input file and expects each record in the work file to produce at most one output record. The template to be used is shown in Figure 36.

ACCESS PATH INFORMATION

access-path-file-name: EMP-HAIR-CMD

variable-file-name: EMP-HAIR-VAR

file-type: generates one record per input value

record length: 10 bytes

fields in file:

emp-id: integer, 5 bytes long, offset 0

emp-hair: character, 5 bytes long, offset 5

INPUT-FIELD FOR MODULE: none

OUTPUT-FIELD FOR MODULE: emp-id, location is work-file

OPERATIONS TO PERFORM:

equal bytes

operand 1 is emp-hair

operand 2 is a constant named MODULE-CONSTANT1

BUFFERS TO ALLOCATE

output file: 1

work file: 1

TEMPLATE: A

Module Descriptor for Module AM-1
FIGURE 35

The action of Template A is to read a record from the work file, then do the operations specified in the module descriptor. If the WR-FLAG has been set as a result of the comparisons performed by DOP, the output record is written to the output buffer. If the WR-FLAG is not set, then the corresponding value for the output record is not

moved to the output file, and the record is effectively removed from the stream. When the end of the file has been reached, do any final operations, write the last buffer to the output file, and close all the files. This template contains meta-commands on lines marked by '+'. As stated previously, these commands allow the template to be tailored to the specific module being processed.

```
+    INI             allocate buffers and open files
+    GNW             get the first record from the work file
     WHILE M1 0      while there are records in work file
     EQ C-EOF 0
+       DOP                   do module operations
        EQ W-FLAG TRUE          check if write flag is set
        IF M2 M2                if flag is set
+         WRO                     write output record
        M2                    end if
+       GNW                     get next work record
     M1                 end while
+    DFO                do final operations
+    WLB                write last buffer
+    CLO
     RET 0 0
```

Template A
FIGURE 36

The first meta-command is INI -- file initialization. This causes a PUSH command with the label INI, to be added to the expanded code file. When the code is executing this will cause control to be passed to the command following the label INI. The commands which do buffer allocation are added to the initialization stack when the first file command for the input, work and output files are encountered. When the processing of the template is

complete, the label INI is added to the expanded code file, and then the buffer allocation commands are added, followed by a POP command.

The next line of the template contains another meta-command, GNW -- Get Next Work Record. The file which contains the code for the access method is named in the module descriptor. A PUSH command with the label GNW is added to the expanded code file, and GNW is added to the subroutine table for later processing. Since this is the first file command for the work file, a command to allocate buffers for the work file is added to the initialization stack. The number of buffers to allocate for the work file is found in the module descriptor; in this case a single buffer is allocated. A flag is set to indicate the commands for buffer allocation for the work file have been added to the initialization stack. When the template is complete, the code for the access method will be added to the expanded code file, followed by a POP command. At this point the initialization stack contains the following command:

```
ALLOC 1 w-buf
```

Lines 3 and 4 do not contain meta-commands, so they are added to the expanded code file "as is". Figure 37 shows the expanded code file after line 4 of the template has been added.

```
1. PSH INI
2. PSH GNW
3. WHILE M1 0
4. EQ C-EOF 0
```

Expanded code file
FIGURE 37

The next meta-command is DOP -- Do Operations. The module descriptor lists only one operation, an equality test. EQUAL has two operands. The first operand is the field emp-hair which is located in the work-file. The field has an offset of 5 bytes from the beginning of the work record and a length of 5 bytes. The second operand is a constant. This means it is stored in the symbol table of the File Access Language program under the name MODULE-CONSTANT1 with a length of 5 bytes.

The code expander adds a PUSH with the label DOP to the expanded code file, and adds DOP to the subroutine table. When DOP is expanded, the code to get the first operand is added. In this case, the first operand is located in the work record. A variable called w-ptr points to the beginning of the work record. The offset of the operand is found, and added to the w-ptr. All arithmetic commands place the answer in the register a-result, so this is moved to the field pointer variable, f-ptr. Next, since the first operand is not a constant, the address is moved to a variable named COP1. The second

```
add w-ptr 5
mov a-result f_ptr
mov f-ptr COP1
```

Code to get first operand
FIGURE 38

operand is a constant, so the address does not need to be computed. Since this is a string comparison, and not a numeric comparison, the number of bytes to be compared has to be stored in the register a-length.

```
mov 5 a-length
```

Next, the command to do the comparison is added. Since this is a contraint operation (all comparisons are considered constraint operations), the write flag needs to be set if the constraint is met. The commands to do this are added. Finally a POP command is added to return control to the main program. Figure 39 shows the code which DOP expands to.

The next two template commands are not meta-commands, so they are copied to the expanded code file. Their purpose is to check if the write flag (WR-FLAG) is set in order to determine whether or not to write the record to the output file. The access module descriptor specified an equality test. If this test is successful, (ie the employee's hair color is blond), the write flag is set and the record is written to the output file, otherwise it is

```
          PSH DOP 0
          |
          |
          |
          RET 0 0
      DOP ADD W-PTR 5
          MOV A-RESULT F_PTR
          MOV F-PTR COP1
          MOV 5 A-LENGTH
          EQB COP1 MODULE-CONSTANT1
          MOV FALSE WR-FLAG
          IF L1 L1
            MOV TRUE W-FLAG
          L1
          POP 0 0
```

DOP expanded for Module 1
FIGURE 39

not.

The next meta-command in the template is WRO -- Write Output. Writing output means moving the fields in the input file to the output file, and moving the "output field" to the output file. The output field is the field in the work file which is moved to the output file. There may be more than one output field. In the example, there are no fields in the input file, so only the output field needs to be moved to the output file. The output field is the employee-id. In this example, the output record consists of the employee-id.

This is the first file command for the output file, so a command to allocate buffers for the output file needs to be added to the initialization stack. The module

descriptor specifies one buffer for the output file, so the command

ALLOC 1 o-buf

is added to the initialization stack. A flag is set to indicate that buffer allocation for the output file has been completed.

The WRO meta-command adds one command to the expanded code file, a PUSH with the label WRO. The name WRO is added to the subroutine table. The WRO subroutine contains the following code. The first command in the subroutine is another PUSH. This push calls a subroutine, CHO (see Figure 40) to make sure there is enough room in the buffer for the output record. If there is not enough room, the buffer is written out to the output file, and the output buffer pointer is reset to the beginning of the output buffer. Next, the name of the output field is found in the module descriptor. The length is moved to the register a-length. Since the field is the first one in the record, the w-ptr points to the beginning of the field as well, so ti-ptr is set to w-ptr. (Remember, WOB expects the variables a-length and ti-ptr to contain the length and location of the item to be moved to the output buffer.) Next a PUSH command for the MWB (Move Work Buffer) subroutine is added and finally a POP back to the main program.

The MWB subroutine moves the work buffer pointer to

```
CHO mov o-buf ob-ptr
    skp o-buf or-size
    eq c-error true
    if W1 W2
WLB opn output write
    add ocnt 1
    mov a-result ocnt
    wrt ocnt o-buf
    mov o-buf ob-ptr
    W1
    mov ob-ptr o-buf
    W2
    pop 0 0
```

Code to check output buffer
FIGURE 40

the ti-ptr and then calls the subroutine WOB (Write Output Buffer) which moves the information to the output buffer. After the call, the work buffer pointer is advanced to the next field in the work record. The File Access Machine code generated by WRO is shown in Figure 41.

The next meta-command encountered is GNW. This was also used in line 2. As a result, the subroutine is already included in the expanded code and may simply be called again. The only action which occurs is another PUSH command is added to the expanded code file. Line 12 contains the meta-command DFO -- Do final Operations. DFO first checks the DFO-stack to see if any of the operations in the module need more processing. In this example, none of the operations in the module require additional processing, so the stack is empty. No code is added to the

```
psh CHO 0
mov 5 a-length
mov 0 s-length
psh MWB 0
pop 0 0
MWB
mov w-buf ti-ptr
psh WOB 0
skp w-buf s-length
pop 0 0
WOB
mvb ti-ptr o-buf
skp o-buf a-length
pop 0 0
```

File Access Machine code generated by WRO
FIGURE 41

expanded code file. The next meta-command is WLB -- Write Last Buffer. This command generates a call to the subroutine WLB. Line 14 generates access commands to close the work file and the output file respectively. The Code Generator now expands the subroutine table, and adds the code on the initialization stack. The complete expanded code is shown in Figures 42 and 43.

```
psh INI 0
psh GNW 0
while M1 0
 eq c-eof 0
 psh DOP 0
 eq w-flag true
 if M2 M2
   psh WRO
 M2
 psh gnw 0
M1
psh WLB
cls work 0
cls output 0
ret 0 0
```

Main loop for Template A
FIGURE 42

```
INI alloc 1 w-buf
    alloc 1 o-buf
    pop 0 0
GNW -- code for access path inserted here --
DOP add w-ptr 5
    mov a-result f-ptr
    mov f-ptr copl
    mov 1 a-length
    eqb copl MODULE-CONSTANT1
    mov false wr-flag
    if L1 L1
     mov true wr-flag
    L1
    pop 0 0
WRO psh CHO 0
    mov 5 a-length
    mov 0 s-length
    psh MWB 0
    pop 0 0
MWB mov w-buf ti-ptr
    psh WOB 0
    skp w-buf s-length
    pop 0 0
WOB mvb ti-ptr o-buf
    skp o-buf a-length
    pop 0 0
CHO mov o-buf ob-ptr
    skp o-buf or-size
    eq c-error true
    if W1 W2
WLB opn output write
    add ocnt 1
    mov a-result ocnt
    wrt ocnt o-buf
    mov o-buf ob-ptr
    W1
    mov ob-ptr o-buf
    W2
    pop 0 0
```

Expanded meta-commands for access module AM-1
FIGURE 43

CONCLUSION

In Chapter 1 we saw various approaches to query optimization. All these approaches were limited because they could only deal with file organizations which the designers had anticipated. Furthermore, we also saw that no single optimization strategy would produce an optimium solution for arbitrary queries. This chapter also discussed the concept of dataflow as an approach to query processing. Using a dataflow metaphor has the advantage of making parallel processing relatively easy to implement because it is trivial to decide what modules can be processed concurrently.

In Chapter 2 dataflow graphs were proposed as a vehicle for describing queries. A dataflow graph is a directed graph with nodes represented as modules. Incoming arcs represent data received by a module and processed by the module. Outgoing arcs represent data produced by the module. This approach has the following advantages:

(1) modules which can be processed in parallel are easily identified. If multiple processors are available, it is easy to use them;

(2) file access can be treated as a virtual operator.

The dataflow graph is implemented using Access Modules. An access module contains at most one file

access and any constraints or computations which can be applied in the module. Templates are used to describe module processing. A template is written in Data Flow Language. Data Flow Language uses Meta-commands to represent I/O specific processing as generic actions.

Chapter 3 described the File Access Machine. This is a processor for implementing dataflow graphs. It is intended as a virtual machine specification for a file processing machine. The File Access Language is a set of primitive instructions which can be used to describe any kind of file access methods.

Chapter 4 showed how the modules of a dataflow graph are implemented on the File Access Machine. Access modules provide specific I/O information for processing a query. Code for processing a module is developed by a code expander using the access module descriptor and a template to produce code in the File Access Language which can be executed on the File Access Machine to provide the answer to the query.

This thesis has shown that if we are given a description of a query solution, specifying only the names of the files to process and the order of processing, we can produce code to solve the query. The code will be tailored to handle file access and I/O dependent processing. This tool can be useful in developing query optimizers which can deal with an arbitrary number of file organizations.

BIBLIOGRAPHY

Ackerman, William B. Dataflow Languages. in AFIPS Conference Proceedings , 1979, 48 , pp 1087-1095.

Astrahan, M.M. and D.D. Chamberlin, Implementation of a Structured English Query Language. Communications of the ACM , 1975, 18,10, pp 580-588.

Astrahan, M.M. et al. System R: Relational Approach to Database Management. ACM Transactions on Database Systems , 1976 , 1,2 , pp 97-137.

Blasgen, M.W. and K.P. Eswaren, Storage and access in relational data bases. IBM Systems Journal , 1977, 16,4 , pp 363-377.

Buneman, Peter, Robert E. Frankel and Risheyur Nikhil, An Implementation Technique for Database Query Languages. ACM Transactions on Database Systems , 1982 , 7,2 , pp 164-186.

Chamberlin, D.D. and R.F. Boyce SEQUEL: A Structured English Query Language. in ACM/SIGMOD Workshop on Data Description, Access and Control, 1974.

Gostelov, Kim P., and Robert E. Thomas, A View of Dataflow. in Proceedings AFIPS 1979 National Computer Conference , 1979 , pp 629-636.

Cardenas, Alfonso F. Data Base Management Systems. Allyn and Bacon, Inc., 1979, Boston.

Codd, E.F. A relational model of data for large shared data banks. Communications of the ACM , 1970 , 13,6 , pp 377-387.

Date, C.J. An Introduction to Database Systems. 2nd Ed., Addison-Wesley Publishing Company, 1977.

Dennis, Jack B. and David P. Misunas, A Preliminary Architecture for a Basic Data-Flow Processor. in 2nd Annual

Symposium on Computer Architecture, 1975, pp 126-140.

Hawthorne, Paula and Michael Stonebraker, Performance Analysis of a Relational Data Base Management System. ACM/SIGMOD 1979 International Conference on Management of Data , 1979 , pp 80-92.

Landin, P.J. A correspondence Between ALGOL and Church's Lambda Notation: Part I. Communications of the ACM , 1965 8,2 , pp 89-101.

Lorie, Raymond A. and Jorgen F. Nilsson An Access Specification Language for a Relation Data Base System. IBM Journal of Research and Development , 1979, 23,3 , pp 286,298.

Selinger, P. Griffiths, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie and T.G. Price, Access Path Selection in a Relational Database Management System. in ACM/SIGMOD 1979 International Conference on Management of Data , 1979 pp 23-24.

Senko, M.E. Data Structures and Accessing in Data-Base Systems. IBM Systems Journal , 1976 , 12,1 , pp 30-93.

Stonebraker, Michael, Eugene Wong, Peter Kreps, and Gerald Held, Design and Implementation of Ingres. ACM Transactions on Database Systems , 1976 1,3 pp 189-222.

Treleaven, Philip C., David R. Brownbridge and Richard P. Hopkins, Data-driven and Demand-driven Computer Architecture. Computing Surveys , 1982, 14,1 , pp 93-143.

Wong, E. and Karel Youssefi, Decomposition - a strategy for query processing. ACM Transactions on Database Systems , 1976, 1,3 , pp 223-41.

Yao, S. Bing Optimization of Query Evaluation Algorithms. ACM Transactions on Database Systems , 1979, 4,2 , pp 133-155.

APPENDICES

APPENDIX A
FILE ACCESS LANGUAGE COMMANDS

1. I/O COMMANDS

These commands open and close files, and read and write files. The variable c eof is set to true when a file is at end of file, and the variable c-error is set if some error occurs during file I/O. When a file is read, a block of the file is read into an internal (to the access machine) buffer. Each buffer has a buffer pointer structure which contains the address of the buffer, and a buffer pointer, which is set to point to the first position of the buffer when the buffer is filled. The buffer pointer can be set to point to other positions in the buffer with the skip command. This buffer remains in use until it is cleared by the user.

OPEN

Open checks the open file array to see if the file is already open. If it is not, the file is opened and the current-file descriptor, c-fd, is set to the file descriptor for the file. If the file is already open, the file array will contain the file descriptor for the file, and the current-file descriptor will be set from the file array. All reads and writes use the file indicated by the current file descriptor.

Opcode: OPN
arg1: name of file (character string)
arg2: mode (integer)

CLOSE

Close closes the file. If this is the last user of the file, the file name is removed from the open file array, otherwise the user count is decremented by one.

Opcode: CLS
arg1: name of file (character string)
arg2: unused

COPY

Copy first gets an empty buffer, and then reads the next sequential block of the file whose file descriptor is in the register c-fd into the buffer. The address of the buffer structure is returned. The register c-eof is set if the end of the file is reached. If the read is unsuccessful for any other reason, c-error is set.

Opcode: CPY
arg1: variable for the buffer structure address
arg2: not used

GET

Get first gets an empty buffer, and then reads the specified block from the file whose file descriptor is contained in the register c-fd into the buffer. The address of the buffer structure is returned. If the read is unsuccessful, the register c-error is set.

Opcode: GET
arg1: offset of block from beginning of file
arg2: variable for returned address

ALLOCATE

Allocate removes an empty buffer from the queue and returns the address of the buffer structure.

Opcode: ALL
arg1: variable for returned address
arg2: not used

CLEAR

Clear resets buffer variables and returns the buffer to the buffer queue.

Opcode: CLR
arg1: address of buffer structure
arg2: not used

WRITE

Write writes the buffer into the current file descriptor. The buffer written is the block indicated by the first argument.

Opcode: WRT
arg1: offset in blocks from the beginning of the file
arg2: address of the buffer structure

SKIP

Skip moves the buffer pointer the specified number of positions. A negative number moves the pointer backward. If an attempt is made to move the pointer past the end of the buffer, or past the beginning of the buffer, c-error is set and the buffer pointer is not moved.

Opcode: SKP
arg1: number of positions to move pointer
arg2: address of buffer pointer structure

2. ARITHMETIC COMMANDS

The access machine supports integer arithmetic. The following operations are supported: add, subtract, multiply, divide and modulo. All the arithmetic commands place the result of the operation in the register _a-result_. If the type of an argument is not an integer, the register _c-error_ is set.

ADD

Add adds the value of the first argument to the value of the second argument and places the sum in the register a-result.

Opcode: ADD
arg1: number or variable
arg2: number of variable

DIVIDE

Divide divides the value of the first argument by the value of the second argument and places the quotient in the register a-result. The remainder is dropped.

Opcode: DIV
arg1: dividend, may be integer or variable
arg2: divisor, may be integer or variable

MULTIPLY

Multiply multiplies the value of the first argument by the value of the second argument, and places the product in the register a-result.

Opcode: MUL
arg1: integer or variable
arg2: integer or variable

SUBTRACT

Subtract subtracts the second argument from the first argument and places the difference in the register a-result.

Opcode: SUB
arg1: integer or variable
arg2: integer variable

MODULO

Modulo divides the value of the first argument by the value of the second argument and places the remainder in the register a-result.

Opcode: MOD
arg1: integer or variable
arg2: integer or variable

3. CONDITIONAL TESTS

The conditional tests are equal, less than and greater than. There are two versons of each test, one for integers, and the other for bytes. If the test is true, the variable c-code is set to one, otherwise it is set to zero. For the byte tests, the number of bytes to be compared is indicated by the register a-length.

EQUAL

The two versions of equal are EQ for integer comparison and EQB for byte comparison.

Opcode: EQ
arg1: integer or variable
arg2: integer or variable

Opcode: EQB
arg1: address of string1
arg2: address of string2

GREATER THAN

The two versions of greater than are GT for integers, and GTB for bytes.

Opcode: GT
arg1: integer or variable
arg2: integer or variable

Opcode: GTB
arg1: integer or variable address
arg2: integer or variable address

LESS THAN

The two versions of less than are LT for integers and LTB for bytes.

Opcode: LT
arg1: integer or variable
arg2: integer or variable

Opcode: LTB
arg1: integer or variable address
arg2: integer or variable address

4. MOVEMENT

MOVE

Move also has two versions, one for integers, and one for bytes. In general, the first argument is the source, and the second is the destination. For an integer move, the value of the source is moved to the address of the destination. For byte moves, the variable a-length indicates how many bytes are supposed to be moved. The first argument contains the address of the source and the second argument contains the destination address.

Opcode: MOV
arg1: source
arg2: destination

Opcode: MVB
arg1: address of source
arg2: address of destination

5. LOGICAL COMMANDS

AND

And takes no arguments. The two commands following AND are evaluated. If they are both true, the register c-code is set to one, otherwise c-code is set to zero. In either case, control is passed to the third command after the AND command.

Opcode: AND
arg1: not used
arg2: not used

OR

Or takes no arguments. The two commands following the OR command are evaluated. If either or both of the commands are true c-code is set to one, otherwise it is set to zero. In either case, control is passed to the third command after the OR command.

Opcode: OR
arg1: not used
arg2: not used

WHILE

While interrupts the sequential processing of the access machine. The first argument of the WHILE command is a label, (it may be a valid opcode, or a marker) which indicates the scope of the while loop. The command following the while command is the "test" for the while loop. This command is executed, and if the variable c-code is true after execution, processing will continue until a command or marker is found which matches the first argument of the WHILE command. At this point, control is passed back to the "test" for the loop. The command is executed, and if c-code is true, another iteration of the loop occurs. Execution of the code inside the loop continues until the "test" command sets c-code to false. When c-code is false, control is passed to the label named as the first argument. If the label is not a valid opcode, processing simply continues with the next command.
Opcode: WHILE

arg1: label
arg2: not used

IF

If tests the value of the variable c-code. If takes two arguments, both labels. The labels indicate which code to process when c-code is true, and which code to process when c-code is false. The first argument is used to indicate where control should be passed when c-code is false. When c-code is true, processing continues until a label is found which matches the first argument. Then control is passed to the label which matches the second argument. When c-code is false, control is passed to the label matching the first argument. Processing begins at this point. The code assumes 1) the code for the true condition immediately follows the if statement and 2) the code for the false condition immediately follows the code for the true condition.

Opcode: IF
arg1: label1
arg2: label2

APPENDIX B

ACCESS MACHINE REGISTERS

REGISTER NAME	COMMANDS WHICH USE THE REGISTER
a-result	add, sub, mul, mod, div
a-length	eqb, ltb, gtb, movb
c-code	and, eq, eqb, gt, gtb, if lt, ltb, or, while
c-eof	copy, get
c-error	alloc, close, copy, get, mov movb, open, skip, write
c-fd	get, copy, write, open, close

APPENDIX C

```
1.  INI             allocate buffers and open files
2.  GNW             get first record from the work file
3.  WHILE M1 0      while there are records in work file
4.  EQ C-EOF 0
5.     DOP                    do module operations
6.     EQ WR-FLAG TRUE          check write flag
7.     IF M2 M2               if write flag is set
8.       WRO                    write output record
9.     M2                     end if
10.    GNW                     get next work record
11. M1                  end while
12. DFO                 do final operations
13. WLB                 write last buffer
14. CLO
16. RET 0 0
```

TEMPLATE A

```
1.  INI       allocate buffers and open files
2.  GNI       get the first record from the input file
3.  WHILE M1 0   while records are in the input file
4.  EQ C-EOF 0
5.    GIF          move input field addr to in-fld-ptr
6.    GNW          get a record from the work file
7.    EQ C-EOF 0
8.    IF M2 M2     if a record is found
9.     DOP                do module operations
10.     EQ WR-FLAG TRUE      check write flag
11.         IF M3 M3        if flag is set
12.             WRO                write output record
13.         M3                  end if
14.   M2           end if
15.   GNI            get next input record
16. M1             end while
17. DFO            do final operations
18. WLB            write last buffer
19. CLO
20. RET 0 0
```

TEMPLATE B

```
1.  INI          allocate buffers and open files
2.  GNI          get the first record in the input file
3.  WHILE M1 0   while there are records in input file
4.  EQ C-EOF 0
5.    GIF          move input field addr to in-fld-ptr
6.    GNW          get a record from the work file
7.       WHILE M2 0  while records are in work file
8.       EQ C-EOF 0
9.         DOP             do module operations
10.         EQ W-FLAG TRUE check write flag
11.         IF M4 M4       if flag is set
12.           WRO            write output record
13.         M4             end if
14.        GNW             get next input record
15.      M2           end while
16.   GNI             get next input record
17. M1           end while
18. DFO          do final operations
19. WLB          write last buffer
20. CLO
21. RET 0 0
```

TEMPLATE C

APPENDIX D

MODULE DESCRIPTION FOR MODULE AM-2

ACCESS PATH INFORMATION

access-path-file-name: EMP-SEX-CMD

variable-file-name: EMP-SEX-VAR

file-type: generates one record

record length: 6 bytes

fields in file:

emp-id: integer, 5 bytes long, offset 0

emp-sex: character, 1 byte long, offset 5

INPUT-FIELD FOR MODULE: emp-id

OUTPUT-FIELD FOR MODULE: emp-id, in work-file

OPERATIONS TO PERFORM:

equal

operand 1 is emp-id

operand 2 is a constant named MODULE-CONSTANT1

BUFFERS TO ALLOCATE

output file: 1

work file: 1

TEMPLATE: B

APPENDIX D

MODULE DESCRIPTION FOR MODULE AM-3

ACCESS PATH INFORMATION

access-path-file-name: EMP-NAME-CMD

variable-file-name: EMP-NAME-VAR

file-type: generates one record

record length: 35 bytes

fields in file:

emp-id: integer, 5 bytes long, offset 0

emp-name: character-string, 30 byte long, offset 5

INPUT-FIELD FOR MODULE: emp-id

OUTPUT-FIELD FOR MODULE: emp-name, in work-file

OPERATIONS TO PERFORM: none

BUFFERS TO ALLOCATE

output file: 1

work file: 1

TEMPLATE: B