AD-A049 804 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
IMPLEMENTATION OF A QUERY LANGUAGE BASED ON THE RELATIONAL CALC--ETC(U)
DEC 76 F R GREEN N00014-75-C-0612
UNCLASSIFIED T-39 NL

| OF |
AD
A049804

END
DATE
FILMED
3-78
DDC

REPORT T-36 DECEMBER, 1976

CSL COORDINATED SCIENCE LABORATORY

IMPLEMENTATION OF A QUERY LANGUAGE BASED ON THE RELATIONAL CALCULUS

UNIVERSITY OF ILLINOIS - URBANA, ILLINOIS

6 IMPLEMENTATION OF A QUERY LANGUAGE BASED ON THE RELATIONAL CALCULUS,

by

10 Fred Randolph Green, Jr.

12 56p.

11 Dec 76

14 T-39

9 Master's thesis,

This work was supported in part by the Office of Naval Research under Contract N00014-75-C-0612

15

097700

IMPLEMENTATION OF A QUERY LANGUAGE BASED ON THE RELATIONAL CALCULUS

BY

FRED RANDOLPH GREEN, JR.
B.S., Virginia Polytechnic Institute and State University, 1970
M.S., University of Florida, 1973

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1976

Thesis Adviser: Professor David L. Waltz

Urbana, Illinois

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. D. L. Waltz, for suggesting this project, Tim Finin and Paul Rutter for their help with the MACLISP system, Woody Conrad for his help with the 3-M data base, and Tze-Wah Wong for providing some LAP code functions for I/O and string handling. I would also like to thank my wife, Carol, for her help in typing and proof-reading this thesis.

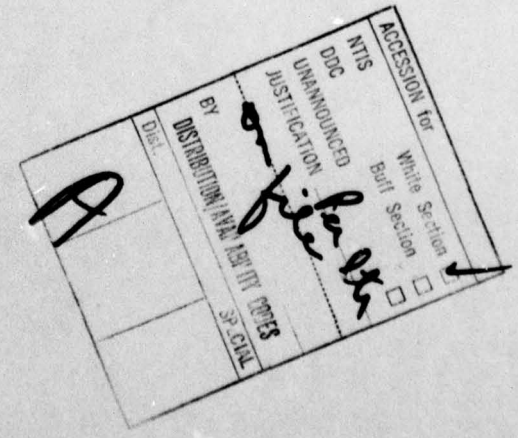

TABLE OF CONTENTS

CHAPTER I

INTRODUCTION -- THE RELATIONAL VIEW OF DATA

The problem with which we have been faced is the development of a query language for the Navy's 3-M data base. The intended application of this language is the intermediate language for the PLANES natural language system. [Waltz, 1976].

Codd [1970] has introduced the notion of a relational view of data. This Data Model (DM) is discussed in detail by Codd [1973a] and Date [1975]. The relational model has its foundation in the mathematical theory of relations. In this model the data is viewed as being divided into "relations" which correspond to files in conventional data base terminology. Each relation contains a collection of "tuples" which correspond to records, and each tuple contains one or more "domains" or fields. A relation can be conveniently viewed as a table with each row being a tuple and each column a domain.

Date regards the relational view as superior to both the hierarchical view of data (typified by the Information Management System (IMS) [IBM, 1971]) and the network view (typified by the Data Base Task Group System (DBTG) [ACM, 1969]) in that the latter two systems are both data dependent while the relational view stresses data independence which means that the user is isolated from the actual physical organization of the data. Data independence is particularly important in this project where we are currently working with a small subset of the large 3-M data base. If our natural language system were to be upgraded to work with the entire data base

(or a larger part of it), substantial changes would be necessary in accessing methods and data organization, but, using the relational model, these changes need not affect the "data model" seen by the users. In addition, the internal tabular format of the data base as it is now organized is ideally suited to a relational model of the data. For these reasons, we have decided to use a relational model for the data base.

A Relational View of the 3-M Data Base

The 3-M data base [OPNAVINST 4790.2A and FMSOINST 4790.1A] is a large data base containing information related to the maintenance and operation of approximately 5000 aircraft belonging to the United States Navy. The data is stored in the data base as card images. Each card image contains a number of domains, the domains present on a given card being determined by the "card type" of the card which is, in turn, determined by the source of the data for the card image. For example, all cards of card type 76 contain daily summaries of flights, and all cards of card type 79 contain monthly summaries of flight and maintenance activity for an aircraft. For the development of the PLANES System we are using a relatively small subset of the 3-M data base which consists of data for 48 selected planes with the data for each plane covering a two year period.

Having chosen the relational model to represent the data, we were then faced with the problem of applying this model to our subset of the 3-M data base. A study of the card types used in the subset data base showed that certain groups of card types could easily be viewed as relations. The card types thus chosen to comprise each relation are shown in Table I

Table I. Relations for the 3-M Data Base Subset

Relation	Card types	Description
A	76	Flight data summary
F	12,32	Failed parts
I	17,27,47	Installed parts
M	11,21,31,41	Authorization of maintenance action
O	79	Overall summary
R	16,26,46	Removed parts
S	[77]	Summary by system
U	71	Maintenance data summary
W	[78]	Summary by work unit code

(card types 77 and 78 are enclosed in brackets since they are not part of the full 3-M data base).

In this manner we have organized the data base into a number of relations with each relation containing certain card types. The tuples in a relation now become the individual card images comprising the data.

To make file accessing more efficient, we have partially inverted the files on two frequently accessed domains, the BUSER (bureau serial number of the individual aircraft) and the ACTDATEYR (year field of the action date).

An important aspect of the file structure of a relational data base is normalization [Codd, 1971a and Date, 1975]. Relations which are normalized are more convenient to manipulate, particularly with respect to updating the data base. The relations we have just defined are normalized in the sense that each domain of a tuple in one of the relations contains a single value, not a set of values. A relation which is normalized in this sense is said to be in the first normal form. There are two other normal forms, second normal form and third normal form, each of which places successively more stringent requirements on the contents of a relation (i.e., a relation in third normal form is also in first and second normal form). We shall not discuss these latter two forms in more detail except to note that our relations only satisfy the requirements for first normal form. While the use of the second or third normal form would provide little advantage over the first normal form for our small, static data base subset, these normal forms could be of considerable value if used in structuring the full 3-M data base which is both large and dynamic.

CHAPTER II

THE DATA SUBLANGUAGE ALPHA

Having selected a model for the data, our next step was to select a query language. Two families of higher level data sublanguages for a relational data base are based on, alternatively, the relational algebra (derived from the algebra of sets) or the relational calculus [Codd, 1971b, 1971c and Date, 1975] (derived from the predicate calculus). Codd [1971c] and Date [1975] have compared these two and found the relational calculus to be superior, particularly for use as a target language for a natural language system. The main reason for their choice of the relational calculus as a target language was that the calculus is non-procedural; i.e., a query in the relational calculus conveys little information about how to proceed in searching the data base. The relational algebra is more procedural which makes the automatic construction of a query by a natural language system somewhat more difficult.

A data sublanguage (DSL) refers to the parts of a query language which are oriented strictly towards data accessing as opposed to computation. The host language (e.g., LISP, SAIL, etc.) is the language in which the DSL is embedded. We have chosen to use the relational calculus sublanguage, DSL Alpha, proposed by Codd [1971b]. For our purposes, the DSL is embedded in LISP, so we have modified the syntax of DSL Alpha (as presented in [Codd, 1974]) to be compatible with LISP. The syntax of the resulting language is presented in Table II (note that, since our natural language system is aimed only at retrieval of data, we omitted those portions of DSL Alpha

Table II. Syntax for DSL Alpha Implementation

```
query-stt :: = (FIND quota (range-specs) (target-list)
                (bool-expr) (sort-expr))
quota :: = integer | ALL
range-specs :: = (tuple-var rel-name) range-specs | null
target-list :: = fterm | fterm target-list
fterm :: = term | (fcn arglist)
term :: = (tuple-var attribute)
attribute :: = domain | (user-fcn)
arglist :: = term | arglist term
bool-expr :: = bool-expr | (quant tuple-var bool-expr) |
                (log bool-expr bool-expr) | (NOT bool-expr) | pexpr
quant :: = ALL | SOME
pexpr :: = (pred jterm jterm)
pred :: = NEQ | EQU | LEQ | GEQ | LT | GT
log :: = AND | OR
jterm :: = term | const
fcn :: = SUM | AVG | COUNT | MIN | MAX
sort-expr :: = (attribute seq) | NIL
seq :: = UP | DOWN
```

which are relevant to data modification). The notation used is BNF; terminal symbols are indicated by capital letters.

The semantics corresponding to this syntax are, briefly, as follows.

1) The function FIND is the top level function of this implementation. It returns an output relation which is a list of the form (⟨name list⟩ ⟨tuple list⟩).

2) The ⟨name list⟩ is a list of domains in the ⟨tuple list⟩.

3) The ⟨tuple list⟩ is a list of output tuples with each tuple in the form given by the ⟨target-list⟩ specification. The ⟨tuple list⟩ contains only unique tuples; i.e., all duplicates are eliminated.

4) ⟨quota⟩ is the maximum number of tuples permitted in the ⟨tuple list⟩ (if more occur, the list is truncated after sorting).

5) ⟨range-specs⟩ associates tuple variables with specific relations (a tuple variable is a variable which takes as values the tuples in its associated relation). All tuple variables to be used in the FIND function must be declared in ⟨range-specs⟩.

6) ⟨target-list⟩ specifies the ⟨fterm⟩'s which make up the tuples in the ⟨tuple list⟩ above.

7) ⟨term⟩ references a specific domain of the relation for the specified tuple variable; i.e., (X HOWMAL) references the "HOWMAL" domain for the variable X. (X (TOTAL)) means that the user defined function TOTAL will be executed for each value of X which permits the summing of several domains.

8) ⟨bool-expr⟩ is a relational calculus expression which must be true for each tuple in the output list.

9) ⟨sort-expr⟩ specifies how the output relation is to be sorted, the domain on which to sort being specified.

We now give two examples of the use of the query language. First we have the query, "Find the total number of hours of unscheduled maintenance for BUSER 3 during 1972." This translates into the DSL Alpha expression:

```
(FIND ALL                                  ⟨quota⟩
      ((V O))                              ⟨range-specs⟩
      ((SUM (V NORMUNS)))                  ⟨target-list⟩
      (AND                                 ⟨bool-expr⟩
       (EQU (V ACTDATEYR) 2)
       (EQU (V BUSER) 3))
      NIL)                                 ⟨sort-expr⟩.
```

Here we have listed nonterminal symbols of the syntax to the right of the corresponding parts of the query. In this query, V is declared to be a tuple variable on the 0 relation (which contains card type 79, monthly summaries). SUM is a built-in function which, in this example, sums the NORMUNS domain over all values of V which satisfy the logic expression. ACTDATEYR is a domain containing the ones digit of the year (this is reasonable since our data base subset spans less than 10 years).

The second example is, "Find the date and not operationally ready hours for all maintenances which were performed on the same day as a flight." Our DSL Alpha expression for this is:

```
(FIND ALL
      ((V1 U) (V2 A))
      ((V1 ACTDATE) (V1 NORHRS))
      (SOME V2 (EQU (V1 ACTDATE) (V2 ACTDATE)))
      (NORHRS DOWN)).
```

In this query V1 and V2 are tuple variables on, respectively, relations U (card type 71, daily maintenance summaries) and A (card type 76, daily flight summaries). In the logic expression (SOME V2 ...), we see that V2 is existentially quantified, so a given value of V1 will satisfy the expression only if there exists a value of V2 such that the ACTDATE domains of the two variables are equal. The sort expression, (NORHRS DOWN), specifies that the tuples in the output relation will be sorted in descending order on the NORHRS domain.

It should be noted that our syntax contains no explicit restriction of the range of a universally quantified variable (e.g., Y in (ALL Y (EQU (X JCN) (Y JCN)))) as does the syntax of alpha expressions as presented by Codd in [1971c]. We have assumed that the range for such a variable is determined by the monadic (i.e., containing a single variable) predicates for that variable in the logic expression. For example, in the expression (ALL V1 (AND (GT (V1 NORHRS) 10) (EQU (V1 JCN) (V2 JCN)))) the universe for V1 would be all members of its associated relation for which the value of the NORHRS domain is greater than 10.

The use of the query language is discussed in more detail in Appendix I, A User's Manual for the DSL Alpha Implementation.

CHAPTER III

REDUCTION TO THE RELATIONAL ALGEBRA

The next important question is how to implement this language. Codd [1971c] proposes an algorithm to reduce the relational calculus to the relational algebra, which, as mentioned above, is more procedurally oriented than the calculus. The operations of the algebra are easily implemented, and so we may search the data base by reducing an expression in the relational calculus to one in the algebra and then executing the latter expression on the data base.

The operations of the relational algebra which are required for the reduction are defined in Table III. Examples of some of these operations are given in Table V using the two relations defined in Table IV. These two relations are relation R with domains A and B and relation S with domain C. The representation of relations is slightly different in Table IV and V; in the notation of Table V, relation R would be {(1,5), (2,5), (1,2), (2,2), (3,2)}.

The reduction algorithm proposed by Codd has the following basic steps.

1) Convert the logic expression to prenex normal form; i.e., separate the quantifiers from the rest of the logic expression leaving a "prefix" containing the quantifiers and a quantifier-free "matrix." The matrix is then converted to disjunctive normal form (i.e., sum of products).

2) For each variable in the query, retrieve the associated relation from the data base, restricting the retrieved relation with the monadic predicates in the matrix.

Table III. Operations in the Relational Algebra

Let R and S be relations, r and s tuples from R and S, respectively, A and B domains of R, and C a domain of S.

Operation	Representation	Definition
Cartesian Product	$R \times S$	$\{(r,s) \mid r \in R \wedge s \in S\}$
Difference	$R - S$	$\{u \mid u \in R \wedge u \notin S\}$
Division	$R[A/C]S$	$\{r \in \bar{A} \mid \forall s \in S[C], (r,s) \in R\}$, $R \subset \bar{A} \times A$, $A \subset C \times \bar{C}$
Intersection	$R \cap S$	$\{u \mid u \in R \wedge u \in S\}$
Projection	$R[A]$	$\{u \mid u \in A \wedge \exists v \bar{A}, (u,v) \in R\}, R \subset A \times \bar{A}$
Restriction	$R[A \, \theta \, B]$	$\{r \mid r \in R \wedge r[A] \theta r[B]\}$, where $\theta \in \{=, <, \leq, >, \geq, \}$
Union	$R \cup S$	$\{u \mid u \in R \vee u \in S\}$

Table IV. Two Examples of Relations

Relation R		Domains	Relation S
A	B		C
1	5		2
2	5		5
1	2		
2	2		
3	2		

Table V. Operations on the Relations of Table IV

$R[B/C]S = \{(1),(2)\}$

$R[A] = \{(1),(2),(3)\}$

$R[B] = \{(2),(5)\}$

$R[A<B] = \{(1,5),(2,5),(1,2)\}$

3) Form the cartesian product of these relations (note that this may require a very large amount of storage space).

4) Form the restriction of the cartesian product by applying the appropriate relational algebra operations; i.e., AND -> intersection, OR -> union, dyadic predicate -> restriction.

5) Apply the quantifiers to the restricted set produced in step 4). This is done by applying the operations of projection for an existential quantifier and division by the universe (i.e., the relation retrieved for the variable in step 2); we assume the universe is not empty) for a universal quantifier. The appropriate operators are applied in the order, from right to left, in which the corresponding quantifiers occur in the prefix.

6) Project the result from step 5) onto the target domains to yield the output relation.

This algorithm suffers from two major weaknesses. First, the algorithm requires an excessive amount of storage. As we noted above, the generation of the cartesian product in step 3) will require a large amount of intermediate storage. Furthermore, all domains of a relation must be retrieved, even though only a few will be used. Second, the process of retrieving relations in step 2) may result in the same tuple being retrieved from the data base more than once. If we are to develop a practical system we must overcome these difficulties, and so we come to the problem of optimization.

Optimization

Since the relational calculus is not a procedural language, it is left to the query system to discover an efficient strategy for searching the data base. While Codd's reduction algorithm provides a relatively straight-forward way to implement a relational calculus language, it is, as we have seen above, an inefficient way to proceed. Palermo [1975] has proposed an algorithm which provides solutions to the two major problems we found in the reduction algorithm. Palermo proposes modifications to the reduction algorithm which minimize intermediate storage requirements and require that no tuple be retrieved more than once. The algorithm uses statistical information about the data base to dynamically determine the order in which the relations in the data base are to be explored (the phase "exploring a relation" is used to indicate a process of extracting information related to a specific query).

In developing his algorithm, Palermo uses the join operation. A join is equivalent to forming a cartesian product of two relations and taking the restriction of the result with a dyadic predicate (i.e., one containing two tuple variables) involving one tuple variable on each of the two input relations. If V1 and V2 are tuple variables on relations R1 and R2 respectively, the join of R1 and R2 with the predicate (Pred (V1 D1) (V2 D2)) is {V1,V2| (Pred (V1 D1) (V2 D2))} where Pred is one of EQU, NEQ, LT, etc. and D1 and D2 are domains in the respective relations. In the reduction from the relational calculus to the relational algebra, a dyadic predicate (with two distinct variables) in the calculus translates into a join in the algebra.

We return to the problem of implementing a join. Since, in general, a join requires the extraction of tuples from two different relations, the join process must proceed in two stages. This is especially true if we wish to ensure that we only retrieve a given tuple from secondary storage once in processing a query. The first stage in constructing a join, then, is the construction of a "semi-join" which consists of pairs of the form (DOMAIN, tuple reference number) where DOMAIN is the value for the tuple of a domain involved in the join.[1] When we explore the other relation in the join we then use the semi-join in an indirect join which produces an indirect relation containing the result of the join. When this indirect join has been completed, we may discard the semi-join as it is no longer needed. In Tables VI and VII we have two sample relations. Here TUPLE# is the tuple reference number. Assume we want to do a join generated by the predicate (EQU (A ACTDATE) (U ACTDATE)) where A and U are tuple variables on, respectively, relations A0002 and U0001. If we explore relation A0002 first, the resulting semi-join would be the same as the relation shown in Table VI. We then explore U0001 and perform the indirect join yielding the indirect relation in Table VIII.

Now, we wish to modify Codd's algorithm to use the indirect join, bearing in mind our goal of minimizing intermediate storage requirements and requiring a given tuple to be retrieved only once from the data base. An algorithm which accomplishes this (Palermo's "Algorithm Using Semi-Joins")

---

[1] This differs slightly from Palermo's definition of a semi-join; in Palermo's semi-join, each value of the domain occurs only once and is followed by a list of tuple reference numbers for that value.

Table VI. The temporary relation A0002

TUPLE#	ACTDATE
1	3092
2	3093
3	3095

Table VII. The temporary relation U0001

TUPLE#	ACTDATE	NORHRS
1	3092	5
2	3092	2
3	3094	16
4	3095	7

Table VIII. An indirect join of relations A0002 and U0001

A0002	U0001
1	1
1	2
3	4

is shown in Fig. 1.

1) First we put the logic expression into prenex normal form with the matrix in disjunctive normal form as was done for Codd's algorithm. We then enter the relation exploration loop where each relation in the data base is searched in turn. This loop involves two distinct processes: selection of a relation and exploration of that relation.

2) In relation selection we use some statistical information about the data base (e.g., number of unique values of a domain) along with information about the query (e.g., which domains are required for each relation and the restrictions on those domains) to estimate the intermediate storage requirements for each unexplored relation. We then select the relation with minimum storage requirements for exploration.

3) The process of "exploring" a relation involves the actual searching of the data base. Before accessing the data base we determine which tuple variables are on the selected relation, and for each of these variables, we construct a modified copy of the logic expression in which only those predicates containing that variable have been retained. We then look at each join term in these logic expressions (the dyadic predicates with two distinct variables), and, if no semi-join has been formed for the term when exploring a previous relation, we will build a semi-join for that term; otherwise, we will execute an indirect join using the existing semi-join.

When the data base is actually searched, we extract those tuples satisfying the monadic predicates in the logic and form the appropriate join or semi-join. Also, for each of these tuples, we extract any domains required for variables in the target list, saving them together with the tuple reference

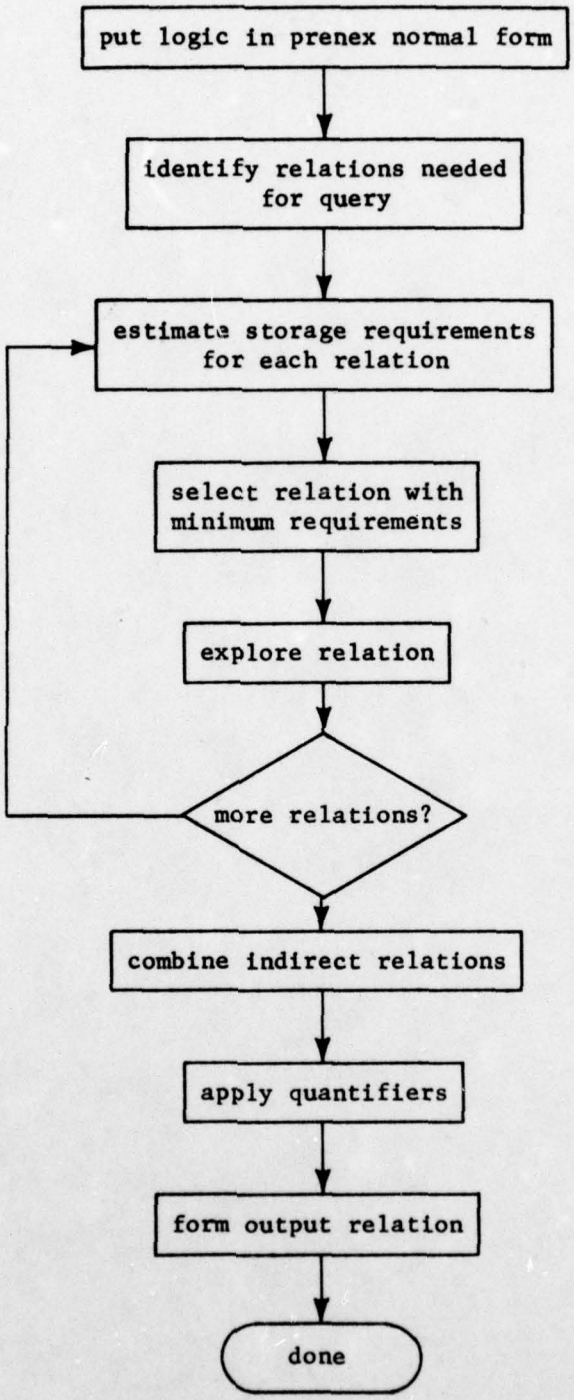

Fig. 1. Palermo's Algorithm Using Semi-Joins

number. For universally quantified variables, we save the tuple reference number to be used later when the quantification operator is applied.

4) When the exploration of a relation is complete, we return to the relation selection. When all relations required for the query have been explored, we combine the indirect relations produced in the indirect joins. We do this by going through the disjuncts of the logic expressions (recall that the matrix is in disjunctive normal form) and, for each disjunct, combining the indirect relations. This combine operation consists of the intersection of two indirect relations on columns representing data base relations common to both indirect relations and the cartesian product of other columns. We then form the union of the indirect relations for the disjuncts. That is, we form an intersection with a cartesian product for two conjuncts or a union for two disjuncts.

5) This resulting indirect relation is then operated on by the quantification operators project and divide as in Codd's algorithm to produce a final indirect relation.

6) Finally, we use this indirect relation together with the domains saved during the relation exploration to construct the output relation with the domains as indicated in the target list, and we are finished.

CHAPTER IV

THE IMPLEMENTATION

We have implemented Codd's DSL Alpha following Palermo's algorithm using semi-joins. A flow chart for the program is shown in Fig. 2. The operation of the program will be discussed below in more detail as we describe the major LISP function in the program and then follow through the execution of two examples.

Our program contains one major enhancement of Palermo's algorithm in our pipelining[2] of single variable queries (which we call "simple" queries) which do not require intermediate data structures to be formed. For these queries, we form the required output relation as the data base is searched rather than building an intermediate object as the data base is searched and then forming the output relation after the search is completed. This method results in substantial savings of time and storage, particularly when a built-in arithmetic function (such as SUM) is being used.

Another feature of this implementation is a capability to save the results of a search for future use. This is important, particularly in a system such as this where a large data base is to be queried interactively. In this situation, interesting results from a query may be expected to elicit another query, and this query is likely to reference the tuples just retrieved. This situation can be recognized in the natural language system

---

[2] By pipelining, we mean a software analogy of the more common hardware term; i.e., the concurrent performance of two or more operations.

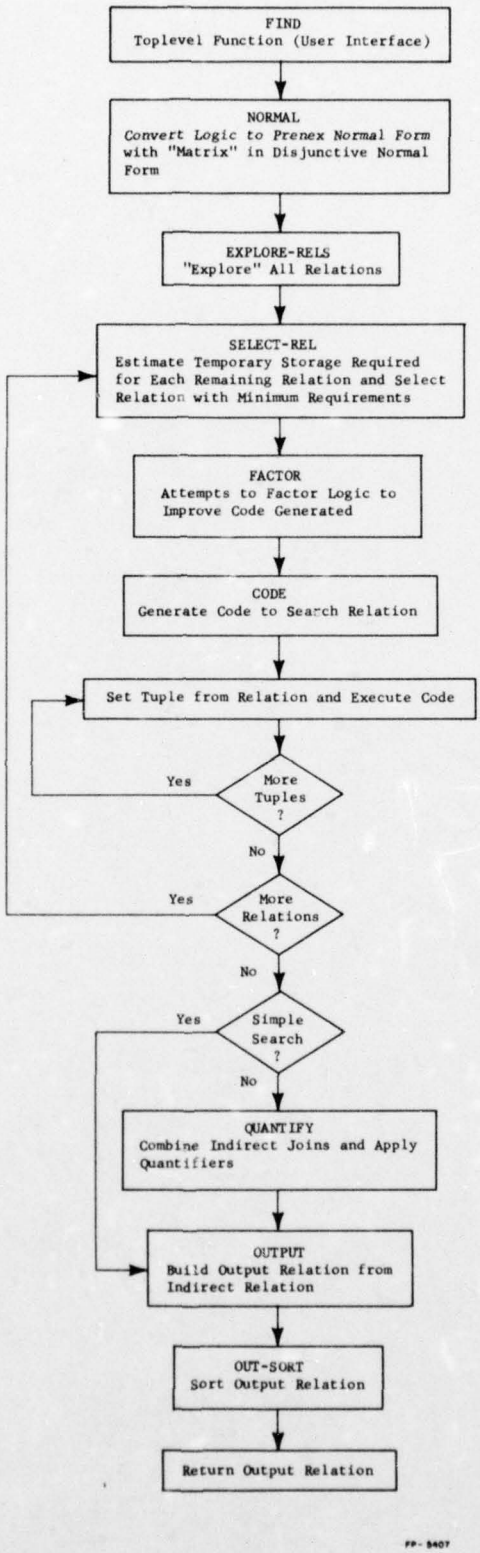

Figure 2. Flow chart of relational database system

when pronoun references are being resolved. To allow search results to be saved, a flag may be set (before the FIND function is called) which will cause all tuples satisfying the logic to be stored in temporary relations. The names of these relations (e.g., U0001) are made available to the user (or the natural language system) who may use them as relation names in subsequent queries.

The Program

The program consists of a number of functions coded in MACLISP [Moon, 1974] which runs on the DEC System 10. The flow of the program through the upper level functions is shown in Fig. 2. In this flow chart, we have indicated the level of a function by a number immediately preceding the function name (i.e., the top level function is number 1, the functions immediately beneath it are 1.1, 1.2, etc.). The top level function is, of course, FIND which is called with the query.

FIND first calls NORMAL which puts the logic expression in prenex normal form with the matrix in disjunctive normal form. The function returns the prefix containing the quantifiers and a list of the disjuncts in the matrix. We then enter EXPLORE-RELS where the relations in the query are explored.

EXPLORE-RELS first calls FACTOR which attempts to factor single predicates out of the matrix. This factoring can substantially improve the speed of the search, particularly if a join term can be factored out of the matrix. We next call the function GENJOIN which attempts to make the disjoints of the matrix consistent with respect to the join operations that each

will produce. This is necessary since some queries contain implicit joins. For example, the query "Which work unit codes always required more than 10 NORHRS?" will generate the query language expression

```
(FIND ALL
      ((V1 U) (V2 U))
      ((V2 WUC))
      (ALL V1 (OR (NEQ (V WUC) (V1 WUC))
                  (GT (V1 NORHRS) 10)))
      (WUC UP)).
```

In the ⟨bool-expr⟩ of this query, only the first disjunct will obviously generate a join operation, but there must also be a join associated with (GT (V1 NORHRS) 10). GENJOIN recognizes this and replaces the second disjunct with (AND (GT (V1 NORHRS) 10) (TRUE (V1 NIL) (V NIL))) where TRUE is a predicate which is always true. This expression will now be recognized as generating a join operation when we generate code to search the data base.

EXPLORE-RELS now builds a list of relations used by the query, and we enter the loop where each of these relations is explored. The relation exploration loop contains two parts: relation selection and exploration of the selected relation.

Relation selection is accomplished by the function SELECT-REL. This function parses the logic expression in a manner identical to that used by the function CODE (see below) in generating the LISP expression which is actually used to explore the selected relation. In SELECT-REL, however, instead of generating code, we employ statistical information on the data base contents (this includes information on the number of unique values of a domain, the range of values for a numeric domain, and the number of tuples in each file in the data base) to generate an estimate of the amount of intermediate

storage which will be required to explore each relation. We then select the relation requiring the minimum amount of storage to explore next. In the case where only one relation remains it is, of course, selected, but the user may always set a flag before calling FIND which will cause an estimate of the number of tuples to be retrieved to be listed for each relation so the user can decide if execution of the query is feasible.

A relation having been selected, we enter EXPLORE for the actual exploration of the relation. Here we first call CODE which does a top down parse of the factored logic expression generating LISP code to be executed against the data base. Code is generated to build any required semi-joins or indirect joins, to construct the universe for universally quantified variables, to retrieve any domains required in the target list, and to save tuples if the results of the search were to be saved (see above). As we noted earlier, CODE recognizes simple (one variable) searches and generates code to build up the output relation directly rather than using an indirect relation.

Two LISP expressions are generated. One is to be executed on the tuples in the data base. The other is passed to GETFILES which selects the files to be searched (we have implemented a simple direct access system by dividing the data base into files according to the values of the domains which are indices in the inverted file structure).

We now enter a loop where the generated code is executed against the tuples in the data base. The function TUPLE gets a tuple from one of the selected files, and we then evaluate the LISP code generated above. This process is repeated until all of the required tuples have been searched.

To improve the efficiency of this search, we have employed functions written in LAP code (MACLISP assembler) to look up files and to read tuples into a buffer; other functions are available to extract domains from the tuples in the buffer.

When all the tuples have been searched, we enter the function CLEANUP. If we have just finished a simple search, CLEANUP puts together an output relation using the results. Having completed CLEANUP, we leave EXPLORE, and, if any relations remain to be explored, we return to SELECT-REL. Otherwise, we leave EXPLORE-RELS and enter QUANTIFY.

In QUANTIFY the indirect relations produced by the indirect joins are combined into one relation using the operations combine and union as we discussed in Palermo's algorithm, and the quantification operations of division and projection are applied. As suggested by Codd [1971c], a check is made to ascertain if the universe is empty before a universal quantifier is applied, and, if it is, the user is notified. For a simple search, QUANTIFY has no effect.

Finally, OUTPUT is called to build the final output relation. For a simple search, a list of output tuples was already built in CLEANUP; however, for a complex search involving indirect relations, it is necessary to build the output relations using the indirect relations and the target list domains which were saved in the data base exploration. Building the output relation for a complex search also necessitates the evaluation of built-in functions in the target list.

When the output relation is complete, it is passed to the function OUT-SORT where the relation is sorted, using the MACLISP SORT function,

according to the sort specifications given in the query.

Finally, we check to see if the number of tuples in the output relation is greater than the quota specified in the query, truncating the relation if necessary. We now return the completed output relation.

Examples

We shall now follow through two examples, one which requires a simple search and the other a "complex" search. These examples are essentially the same as those used above in Chapter II.

Simple Search

Given the query: "Find the total number of hours of unscheduled maintenance for BUSER 3 during 1972," which translates into the DSL Alpha expression:

```
(FIND ALL
      ((V O))
      ((SUM (V NORMUNS)))
      (AND (EQU (V ACTDATEYR) 2)
           (EQU (V BUSER) 3))
      NIL).
```

The top level function FIND first calls the function NORMAL. Since the logic expression is a single conjunct, the expression is returned substantially unchanged. We now enter EXPLORE-RELS in which the actual search is carried out.

FACTOR has no effect since there is only a single disjunct in the logic expression (factoring is frequently necessary in other cases since the process of putting a logic expression into disjunctive normal form often causes the same predicate to appear in every disjunct, and evaluation of an

expression in this form would obviously be very inefficient). Since only one relation, O, is involved, SELECT-RELS simply returns that relation. After SELECT-RELS has been exited, the function CODE is entered.

When CODE is called to generate the code for the search, two important things are discovered: 1) It is realized that both ACTDATEYR and BUSER are index domains and some code is generated which will cause only those tuples in relation O and with ACTDATEYR = 2 and BUSER = 3 to be accessed on disk. 2) It is recognized that this is a simple search, so the code (for the SUM function) is generated to sum the NORMUNS domain for each tuple as it is retrieved. The code generated by 2) is simply:

(STORE(TOT* 0) (PLUS (TOT* 0) (EXTRACT 59 3 'N)))

Here TOT* is an array element (an array is used since there may be more than one function in the target list) which is initialized to zero at the start of the search and EXTRACT is a function which extracts a numeric ('N) field of length 3 (in this case the NORMUNS domain) from a card image (tuple) beginning at column 59. STORE is the standard LISP function which assigns the value of its second argument to the array element in its first argument (in FORTRAN we might write TOT(I) = TOT(I) + EXTRACT(59,3)).

We now exit from CODE and enter a loop where the required tuples are read from disk one at a time with the above expression being executed on each tuple. When the search is completed, we have TOT*(0) = 324.0.

We save this result and exit from EXPLORE-RELS. We now enter the function OUTPUT where, since sorting is not required, we simply return the output relation:

(((SUM (NORMUNS))) (324.0)).

Complex Search

Suppose that we have executed the two following queries, saving the results in temporary relation A0002 and U0001, respectively: 1) Find the daily flight summaries for BUSER 3 for April, 1973; and 2) find the daily maintenance summaries for BUSER 3 for April, 1973. The values of some selected domains of the tuples in these relations might be as shown in Tables VI and VII. We wish to determine the results of the query: "Using the results of the two previous queries, find the date and not operationally ready hours for all maintenances which were performed on the same day as a flight." This query translates into:

```
(FIND ALL
      ((A A0002) (U U0001))
      ((U ACTDATE) (U NORHRS))
      (SOME A (EQU (U ACTDATE) (A ACTDATE)))
      (NORHRS DOWN)).
```

This time, NORMAL removes the quantifier (SOME A) from the logic expression and returns it and the remainder of the logic expression (the "matrix" which is, in this case, (EQU (U ACTDATE) (A ACTDATE))). These are passed to EXPLORE-RELS which first calls FACTOR and GENJOIN, which have no effect in this example. EXPLORE-RELS then builds a list of relations to be explored (A and U), and, finally, calls EXPLORE.

EXPLORE first calls SELECT-REL which estimates the intermediate storage that would be required by each relation if it were explored first and then selects the relation with the minimum storage requirements. The storage requirements are estimated by following through the code generation process used by CODE; but, instead of generating code, statistical information on the data base (e.g., the number of unique values in a given domain of a relation)

is used to estimate the probability that a tuple will satisfy a given predicate. These probabilities are then multiplied by the number of tuples to be searched to derive the storage estimate. In the present case, the only predicate is a join term (see the section "Optimization") which does not restrict the number of tuples to be retrieved (the reason for this is that we first form a semi-join), so we see that U0001 will require more storage than A0002 since U0001 has more tuples and also requires the storage of the target domains. Since we are attempting to minimize storage requirements, SELECT-REL chooses A0002 to explore first.

We now enter CODE. Here, again, the predicate is recognized as a join term which is so named because in the translation from relational calculus to relational algebra this predicate translates into a join operation on the domains in the predicate. Following Palermo, we have employed indirect relations for internal manipulation. As we have discussed above, in an indirect relation, rather than working with an entire tuple, we use only a tuple reference number which uniquely identifies the tuple. In Tables VI and VII, the TUPLE# column is the tuple reference number (these numbers are not part of the permanent relations on disk, but are generated as the tuples are input). For example, relation A0002 can be represented as simply the indirect relation (1 2 3). A join is implemented in two stages using indirect relations. First, the semi-join is formed for the first relation to be explored. A semi-join consists simply of a list of pairs of the form (⟨domain⟩, ⟨tuple reference number⟩) where ⟨domain⟩ is the domain on which the join is to be performed. In our example, the code which would be generated for this is

(SEMIJOIN [address] (EXTRACT 21 4 'N))

where [address] is an address where the results are to be stored and the function EXTRACT gets the ACTDATE domain as described in the example of a simple search. A search of the relation A0002 with this code yields the semi-join:

((3092 1) (2093 2) (3095 3)).

Upon the completion of this semi-join, we return to SELECT-REL where U0001 is found to be the only remaining relation. When CODE is entered, it realizes that A0002 has already been explored and so an indirect join is generated:

(IJOIN [address] [semi-join on A]

(EXTRACT 21 4 'N) 'EQU)

where [address] is as above and [semi-join on A] is the semi-join just created. The result of the indirect join is a list of pairs of tuple reference numbers with the numbers in a pair corresponding to the two variables in the predicate and with each pair satisfying the predicate. The indirect relation produced by the above indirect join is:

((A U) ((1 1) (1 2) (3 4)))

where the variable names are carried along to identify the columns of the indirect relation.

In exploring this relation, one complication occurs which we did not mention above. When CODE was called for this relation, it was also discovered that there were two domains of the variable U to be retrieved for the target list (i.e., ((U ACTDATE) (U NORHRS))). At that time code also was generated to retrieve these domains and the corresponding tuple reference numbers, so this information was also retrieved at the same time that the indirect join was being formed.

Having completed the exploration of both relations, we now exit EXPLORE-RELS and enter the function QUANTIFY. In this function, all indirect relations which have been generated in the data base search are combined into one indirect relation and then suitable operators are applied to perform the quantifications indicated in the logic expression. In our example, only one indirect relation was produced in the relation exploration, so we only need to apply the quantification operation.

The existential and universal quantifiers of the relational calculus correspond (as we saw in Codd's reduction algorithm) to the operations of projection and division, respectively, in the relational algebra. In our example, A is existentially quantified so we need to project our indirect relation onto the variable U. To do this, we simply remove the tuple reference numbers for the A0002 relation and eliminate any duplicates in the resulting indirect relation (in this case there are none). This yields the result:

((U) ((1) (2) (4))) .

We now leave QUANTIFY with this relation and enter OUTPUT where we assemble the output relation by taking those items in the target list with tuple reference numbers corresponding to those in the indirect relation and eliminate any duplicate tuples. This gives us the result:

((ACTDATE NORHRS) ((3092 5) (3092 2) (3095 7))).

This is now sorted on the NORHRS domain to yield the final output relation:

((ACTDATE NORHRS) ((3095 7) (3092 5) (3092 2))).

CHAPTER V

IMPROVEMENTS

While Palermo's algorithm and our implementation of it are both reasonably efficient, a number of improvements are possible. We shall first discuss some shortcomings of our implementation with respect to both data base organization and software and then we shall propose some improvements which are possible in Palermo's algorithm.

One of the areas where the efficiency of our implementation could be substantially improved is by altering the file structure we have utilized for the data base. As discussed earlier, we chose to store our data in card image form with an indexed organization implemented by using index domains as file names. While this has made our implementation easier by allowing the monitor to handle the direct access I/O, it would be more efficient to use a more conventional arrangement where we would maintain our own index files (see, for example, [Date, 1975] for a discussion of indexing techniques). This would also permit direct access on more domains since we would no longer be restricted to the length of a file name for our index domains.

Since the data base contains many numeric domains, much time is now required to extract these domains from the card images and convert the resulting character strings into arithmetic representations. Clearly, a substantial savings in processing time could be achieved by storing these domains in their arithmetic forms.

The final improvement in data base organization which we propose is the conversion of the relations to third normal form which we mentioned earlier. This conversion, which we shall not discuss in detail, involves breaking a relation into smaller relations by taking projections.

The next area where our implementation could be improved is in the data structure used for the storage of intermediate objects in the query (e.g., semi-joins and indirect relations). To simplify free storage management, we chose to use list structures, which are managed by the LISP system, for internal storage. The use of lists, however, is somewhat inefficient, both in storage utilization and execution time, when compared with arrays. In MACLISP, there is one machine word of overhead for each node in a list. While this is not excessive, it requires more than twice as much space to store an indirect relation in a list instead of an array. The building and accessing of lists may also require more execution time than with arrays. In building lists, storage must be allocated to a list as each new object is added to the list, which is not true for arrays. In accessing lists all access is, of course, sequential; for arrays, however, there is no such restriction and techniques such as binary search may be employed. We now use sorting to speed the search of indirect relations (as lists) in the divide operation, and have employed a hashing scheme using arrays of lists to identify indentical tuples (for the project operation, for example). While these techniques have dramatically improved the program performance, it still falls short of what could be done with arrays.

The use of arrays to store intermediate objects would also make it quite simple to move a very large indirect relation to and from secondary

storage. This cannot be done conveniently with lists and so we must rely on the paging system in DEC-10 TOPS monitor to handle this operation for large lists. Using the monitor's paging algorithm degrades performance somewhat itself (see Casey and Osman [1974] for a discussion of paging for a similar application), subjects a larger area of memory to garbage collection, and limits the size of the intermediate objects which can be processed to the address space available to a user. In summary, a significant improvement in performance could be achieved by using array storage with a free storage management system designed for this program.

With the exeception noted above, we have made little use of knowledge about the sort order of the relations we are manipulating. The importance of using this information has been emphasized by Smith and Chang [1975] (see below).

Another possible source of inefficiency is in our decision not to implement the "reduced ranges" proposed by Palermo. A reduced range is obtained by taking the projection of a relation onto domains referenced in the query. The inclusion of reduced ranges might result in some savings in both speed and storage; however, due to the occurrence in this data base of many numeric and other domains with a very large number of possible values,the use of reduced ranges will often result in a relatively small reduction in the number of tuples to be processed. This same argument was employed in deciding to use the definition of a semi-join given earlier rather than Palermo's definition. In spite of this, the use of reduced ranges and Palermo's definition of a semi-join could result in improved efficiency for some queries.

We will now discuss two areas in which improvements may be made in Palermo's semi-join algorithm. These involve the optimization of the query after reduction to the relational algebra and the use of a matrix (from the logic expression) in which the number of dyadic predicates has been minimized.

Palermo [1975] indicates that the efficiency of his algorithm may be improved by applying the quantification operations at an earlier point in his program, but he offers no suggestions as to how this may be done. Smith and Chang [1975], however, discuss the general problem of optimizing a query in the relational algebra. They represent a query as an operator tree and consider the problem of transforming these trees to yield an optimum statement of the query. The ideas used in their "tree transformer" are clearly applicable to our problem of optimizing the placement of the quantification operators. The transformations presented by Smith and Chang appear to suggest no change in the placement of the universal quantifiers (division) in Palermo's algorithm, but could move the existential quantifier (projection) to an earlier point in the relational algebra statement of the query.

Another useful idea presented by Smith and Chang is the "coordinating operator constructor." This system attempts to improve efficiency by coordinating the sort order of domains in the intermediate relations used by the relational algebra operators. Many of these operators can be much more efficient if they operate on a relation with a certain domain in a known sort order. As discussed above we have made some limited use of sort order information, but have not attempted to coordinate sort orders throughout the program.

One area of algorithm optimization which is especially important for natural language applications where the relational calculus expression may, of necessity, be very poorly constructed is the optimization of the logic expression prior to its reduction to the algebra. In particular, we would like to minimize the number of dyadic predicates (which generate joins in the reduction). We have attempted to do this by factoring the logic expression before generating code from it; however, our factoring algorithm only factors out individual predicates and does not do more complicated rearrangements involving conjunctions or disjunctions of predicates. The problem of minimizing a logic expression is a standard one in switching theory and has been discussed extensively in the literature; see, for example, Givone [1970] for a discussion of this topic.

CHAPTER VI

CONCLUSION

In the last chapter, we attempted to point out some areas where substantial improvements could be made in the performance of the program. Assuming that some of the more important improvements had been made (e.g. converting the internal data structures to arrays) we might now consider applying this program to the full 3-M data base. With a data base as large as this, it is not practical to have the entire data base available on direct access devices. As a consequence, the data should be stored with the most recent data and, possibly, some summary data, such as the O relation, on a direct access device with an organization similar to that proposed in the previous chapter. The remainder of the data could be stored on magnetic tape. The accessing of this data would be most efficient if it were stored by relation with the numeric domains encoded in an arithmetic format (so, as pointed out in the previous chapter, data conversions would be minimized). In using magnetic tapes, the program would operate in the same manner as it does now, but the program would have a list of data base tapes, just as it now has a list of files, which it would search to determine which tapes would be required in exploring a given relation. To explore a relation, the program would request that each of the selected tapes be mounted, one at a time, until the relation exploration had been completed.

Our DSL Alpha implementation could be used without the natural language system to provide convenient access to the data base for those users familiar with DSL Alpha, and also to permit any queries not handled by the

natural language system to be used. If this were to be done, a preprocessor should be added to the program to thoroughly check the input query and correct errors in a dialog with the user before beginning actual execution of the query.

In conclusion, we have seen that the relational model provides a good way to view the data in the 3-M data base and that the DSL Alpha provides a data sublanguage which is well suited for use with a natural language system. Our implementation of the DSL Alpha query language currently provides a relationally complete (as defined by Codd [1971c]) interface between the natural language system and the data base. The system also provides features such as the storage of search results and the estimation of the size of search results. We have attempted to analyze the algorithm used for our implementation to discover areas where further improvements in efficiency may be made so that it would be practical to use DSL Alpha with the full 3-M data base.

REFERENCES

Casey, R. G. and Osman, I., "Generalized Page Replacement Algorithms in a Relational Data Base," Proc. ACM-SIGMOD Workshop on Data Description, Access, and Control (Ann Arbor, Michigan, 1974).

CODASYL DBTG Report (ACM, 1969).

Codd, E. F., CACM 13(6), 377 (1970).

Codd, E. F., "Normalized Data Base Structure: A Brief Tutorial," Proc. ACM-SIGFIDET Workshop on Data Description, Access, and Control (San Diego, 1971a).

Codd, E. F., "A Data Base Sublanguage Founded on the Relational Calculus," Proc. ACM-SIGFIDET Workshop on Data Description, Access, and Control (San Diego, 1971b).

Codd, E. F., "Relational Completeness of Data Base Sublanguages," Courant Computer Science Symposium 6, "Data Base Systems" (1971c).

Codd, E. F., "Seven Steps to Rendezvous with the Casual User," IFIP Working Conference on Data Base Management (1974).

Date, C. J., An Introduction to Database Systems (Addison-Wesley, 1975).

FMSOINST 4790.1A, Catalog of 3-M Aviation Information Reports, Dept. of the Navy, (Mechanicsburg, Pa., 1974).

Givone, P. D., Introduction to Switching Circuit Theory (McGraw-Hill, 1970).

Information Management System IMS/360, Application Description Manual (IBM, 1971).

Moon, D. A., MACLISP Reference Manual (Massachusetts Institute of Technology, 1974).

OPNAVINST 4790.1A, "Maintenance Data Collection Subsystem," The Naval Aviation Maintenance Program, Vol. III, Dept. of the Navy (Washington, D.C.).

Palermo, F. P., "A Data Base Search Problem," COINS IV (1975).

Smith, J. M. and Chang, P., CACM, 18(10), 568 (1975).

Waltz, D. L., Naval Research Reviews, 29(1), 11 (1976).

APPENDIX I

USERS MANUAL FOR THE DSL ALPHA IMPLEMENTATION

This manual is intended to provide a user of our DSL Alpha implementation with some practical information on the use and modification of the system. In this appendix, we assume that the user has some knowledge of MACLISP (see Moon [1974]). DSL Alpha and an overview of our implementation of it have been discussed in the main body of the thesis, and we assume here that the user is familiar with the syntax of the DSL as presented in Table II. The only way that the syntax of the actual implementation differs from that presented in the body of the thesis is that the arguments to the function FIND are evaluated when the function is called, meaning that the arguments of FIND should have been quoted (we did not quote them for reasons of clarity). Several examples of actual queries are shown in Table IX.

Operation

This section deals with the basic operation of the program in the MACLISP environment. Before the program can be run, it is necessary to FASLOAD the binary code for the functions used (which are all in compiled form), to read in some tables, and to initialize a few variables. All this is done by reading in the file SEARCH.LSP. It should be noted that BASE and IBASE are changed to 10 (decimal) in this operation, so any PPN's used after the initialization must be expressed in decimal rather than the usual octal.

Table IX. Sample DSL Alpha Queries

1) A query using existential quantification

```
(FIND 'ALL
      ((U U) (A A))
      ((U ACTDATE) (U WUCSYS))
      (SOME A
            (AND (EQU (A ACTDATE) (U ACTDATE))
                 (EQU (A BUSER) 3.)
                 (EQU (U BUSER) 3.)))
      (ACTDATE UP))
```

2) A query using a function as a domain

```
(FIND 'ALL
      ((V O))
      ((V ACTDATE) (V NORMUNS) (V (NOR)))
      (AND (EQU (V BUSER) 3.) (GT (V NORMUNS) 20.))
      (ACTDATE UP))
```

3) The NOR function used in query 2)

```
(DEFUN NOR NIL
      (PLUS (GETFIELD 'NORMSCH)
            (GETFIELD 'NORMUNS)
            (GETFIELD 'NORS)))
```

4) A query using a temporary relation

```
(FIND 'ALL
      '((O 00001))
      '((MAX (O NORMUNS)) (AVG (O NORMUNS)))
      T
      NIL)
```

5) A query using universal quantification

```
(FIND 'ALL
      '((U U) U1 U))
      '((U WUCSYS))
      '(ALL U1 (AND (EQU (U1 BUSER) 3.)
                    (EQU (U BUSER) 3.)
                    (EQU (U1 ACTDATEYR) 3.)
                    (EQU (U ACTDATEYR) 3.)
                    (OR (NEQ (U WUCSYS)
                             (U1 WUCSYS))
                        (GEQ (U1 NORHRS) 10.))))
      '(WUCSYS UP))
```

The project-programmer numbers under which the data files are stored are passed to the program as the values of two atoms. The value of the atom DATAFILESPPN should be a list containing the PPN where the data base files are to be found, and the atom TEMPFILESPPN should contain the PPN where the temporary relations (see below) to be read by the program are stored. Both of these atoms are initialized when SEARCH.LSP is read in; DATAFILESPPN is currently initialized to (512. 140.) and TEMPFILESPPN is initialized to the user's area (as returned by (STATUS UNAME)).

The program uses list structures for storage of intermediate quantities, so it is normal for an occasional garbage collection to occur during program operation. To avoid an excessive number of these, however, it is necessary to give the program an adequate amount of storage space. With the current version of MACLISP (LISP 1137) and with the program running without the natural language system, the storage allocation to MACLISP should be as follows:

CORE	80.	(pages)
FXS	3000	
FLS	2000	
ARRAY	200	

The defaults will do for the other allocations. For complex searches (searches using more than one tuple variable) large internal data structures are built up which may require a larger storage allocation, but the allocation above is a good starting point.

A number of useful options are available through switches which are implemented as atoms which take on the values T or NIL. All switches are set to NIL when SEARCH.LSP is read in. These switches are as follows.

1) SAVE*. When SAVE* is T, all tuples satisfying the logic expression in a query are saved in one or more temporary files. The file name(s) are available as the value of the atom TEMP-FILES upon completion of the FIND function. These file names are of the form Rdddd where R is a relation name and dddd are four digits generated by the GENSYM function. A temporary relation may be referenced in a subsequent query by specifying the file name as a relation (see example 4). Note that TEMP-FILES is initialized to NIL at the beginning of each query where SAVE* is T.

2) ESTIMATE-SIZE. When this is T, the number of tuples to be searched, number of tuples to be retrieved, and storage required (in words) by a query are listed for each relation to be searched. This information may be useful in deciding whether a query will get too many hits to be worthwhile or will require more storage than has been allocated.

3) PRINT-SIZE. This switch causes the actual sizes of the internal lists built during the query execution to be displayed. This is intended primarily for debugging.

4) PRINT-CODE. This causes the code generated to explore each relation in the query to be listed. The code is listed as two S-expressions: the first is the code actually used to search the data base and the second is the code used to select the files to be searched. Note that GRINDEF is loaded to print these expressions so this option should not be used if little free storage space is available.

5) COMPILE-CODE. This allows the data base search to be done using compiled code rather than interpreted code. This is a messy process, however, and its use is not advised unless a large portion of the data base (i.e., several thousand blocks) is to be searched. When this option is specified, a break will occur before each relation is searched. When the break occurs, the file CODE.LSP will exist in the user's area. This should be compiled and assembled using NCOMPL and FASLAP from a separate job. Then type "⟨altmode⟩p" to MACLISP to return from the break and begin the search.

We now turn to some peculiarities of the program of which the user should be aware. First is the use of the ACTDATE domain. Due to the occurrence of at least two formats for the date, ACTDATE is not recognized as a direct access domain and, consequently, its use in predicates can result in unnecessarily long searches. The domain ACTDATEYR provides direct access to files for a specified year and should be used (with ACTDATEMON or ACTDATEDAY, if necessary) in monadic predicates rather than ACTDATE.

The relation returned by FIND contains only unique tuples; however, situations sometimes arise where it is desirable to see which tuples from the data base satisfy the logic expression. This may be done by placing the domain "KEY" in the target list. This domain takes on a unique value for each tuple that is retrieved from the data base and may be used only in the target list (not in the logic expression).

Although it is not clear from the syntax, both terms and built-in functions cannot occur in the same target list; i.e., a target list of the form ((V BUSER) (SUM (V NORHRS))) is not permitted. Also, more than one occurrence of the COUNT function is not permitted in a simple query.

The program does some, though not extensive, error checking as a query is executed. Errors are generally signalled as a break with some indication of the cause of the error. Other errors which can occur are errors from the I/O package which usually occur as a result of a file not being found, and, of course, there are the errors which are detected by the LISP system.

Modifying the Definition of Domains and Relations

In the course of using the program, it may be desirable to change the definitions of some domains or relations. Definitions of the domains used in the data base are stored in the file FIELDS.DEF; similiarly, the relation names and data file names are stored in, respectively, RELS.DEF and FILES.DEF (note that neither of these files contain any information on temporary relations).

We shall start with FIELDS.DEF. This file consists of one line for each domain. Each line is of the form:

⟨domain name⟩ ((⟨card types⟩)) ⟨start column⟩ ⟨end column⟩
⟨field type⟩ ((⟨statistics⟩)) ⟨graph⟩

where ⟨card types⟩ are the card types on which the domain occurs, ⟨start column⟩ and ⟨end column⟩ give the position of the domain on the card image (1 is the first column), and ⟨field type⟩ gives the type of field: N (numeric) or C (character). ⟨statistics⟩ is a list of statistical information used to estimate the number of tuples to be retrieved. This list contains three numbers: 1) (number of distinct values of the domain)$^{-1}$, 2) the minimum value of the domain, and 3) the maximum value of the domain (parts 2) and 3) are, of course, only applicable to numeric domains). ⟨graph⟩ is a code which tells

how the domain is to be displayed in a graph: Y (y-axis), X (x-axis), or N (should not be displayed graphically).

RELS.DEF is a list of sublists. Each of these sublists is of the form (⟨relation name⟩ (⟨list of card types⟩) where the card types are those occurring in the relation. FILES.DEF is another list of sublists. Each of these sublists is of the form (⟨file name⟩ ⟨number of tuples in file⟩). Each ⟨file name⟩ is a six character name of the form: relation (1 character), BUSER (2 characters), PLANETYPE (2 characters), ACTDATEYR (1 character). Notice that the list of file names serves as the index file for our partially inverted file structure. Changes in the domains, relations, or files may be made by changing the appropriate file(s). Some complications arise when changing a direct access domain. This involves changing some or all of FIELDS.DEF, FILES.DEF, and also three functions in the program (CONJUNCTION, CONJUNCTION*, and FILEFIELD) which process these domains.

Before the DSL Alpha implementation can be used, it is necessary to read in the three definition files and, for each domain in FIELDS.DEF, to set up property lists with the properties CARDTYPES, FIELDPOSITION, TYPE, PROB, and GRAPH. This is done by the functions INITFILES AND INITFIELDS both of which are defined and then evaluated in SEARCH.LISP.

Modifying the Program

It may be necessary, at some future date, to modify the program, either to correct an error, to make a small change (as is necessary for altering a direct access domain), or to make an improvement in the algorithm. For whatever reason the modification is made, some knowledge of the structure

of the program is desirable. The source code for the program is stored in the ten files which are listed in Table X along with a description of the contents of each file. An effort has been made to place closely related routines in file together; for example, all the functions used in putting a relational calculus expression in prenex normal form are found in QNORM.LSP. The relationahips between the higher level functions in the program are shown in the flowchart in Fig. 2. All the functions in the implementation are listed by file in the declaration file, QDCL.LSP.

In modifying the program, some of the most serious problems are likely to be in understanding the data structures used for intermediate storage. There are relatively few global (i.e., "special" in MACLISP terminology) variables, and these are all listed in QDCL.LSP. Most of the data structures are lists which are built up on appropriate parts of the actual query; i.e., a semi-join or indirect relation is appended to the dyadic predicate which caused it to be generated; a target domain is appended to the appropriate term in the target list; and the universe for a universally quantified variable is appended to the appropriate quantifier in the prefix of the logic expression. In manipulating these list structures, frequent use is made of the functions RPLACA and RPLACD (which replace the CAR or CDR, respectively, of a list) which efficiently alter a list and also provide the effect of call by reference for function arguments.

In order to obtain reasonable execution times for the DSL Alpha implementation, it is necessary to use compiled LISP code. Consequently, whenever a function has been modified in one of the source files, that file should be recompiled and assembled. To do this for QSRCH.LSP, for example,

Table X. LISP Source Files

File	Contents
QCODE.LSP	Functions to generate code for relation exploration
QDCL.LSP	Declarations for compiling all functions
QEST.LSP	Functions to estimate number of tuples to be retrieved and to select relations to be explored next
QFIND.LSP	Top level function
QIO.LSP	LAP code I/O and string functions
QNORM.LSP	Functions to put logic in prenex normal form
QOUT.LSP	Functions to prepare the output relation
QREAD.LSP	Functions to drive those in QIO.LSP
QSRCH.LSP	Functions to explore relations
QUANT.LSP	Relational algebra functions

the following BATCH job should be executed:

```
.R COM:NCOMPL
*QDCL.LSP
*QSRCH.LSP
.R COM:FASLAP
*QSRCH.LAP
```

Notice that QDCL.LSP must be compiled before any other files (except QIO.LSP, see following) since it contains the declarations necessary for compilation. Since QIO.LSP is actually a LAP file, it does not need to be compiled, only assembled using FASLAP.

Modifying the Data Base

The final problem which we shall discuss is modifying the data base. Since our implementation of DSL Alpha contains no facilities for updating the data base, any changes to be made must be done using other software. Therefore, in order to alter information in the data base, it is necessary to understand the data base structure.

The data base consists of a number of files with the contents of the files and the file names determined by the direct access domains (see the section "Modifying the Definitions of Domains and Relations"). Each file consists of card images with each card being 80 seven-bit ASCII characters (i.e., 16 36-bit words) long. The domains present in each card image are determined by the card type which is a two digit code in the first two columns of a card. All the domains in our subset data base are listed in the file FIELDS.DEF (see the section "Modifying the Definitions of Domains and

Relations") which contains the card types in which each domain appears and the position of the domain in the card image. Note that each card image is exactly 80 characters long and is not followed by a carriage return or any other delimiter.

Should it be necessary to add data to the data base or to replace the data we are now using with new data, programs are available to convert the print tapes on which the Navy sends us data (see [FMSOINST 4790.1A]) into the files for our subset data base. These programs are all written in SAIL.

The most important of these programs is REFORM.SAI which actually reformats the print tape. This program is driven by four tables which contain: 1) the position of the fields in an input record, 2) for each pair (card type, field) the position and length of the field in the output record, 3) a string of letters such that the i'th character in the string is the relation containing card type i, and 4) a hash table for converting BUSER from the six digit Navy BUSER to a two digit code. In 1) and 2), the fields are identified by integers rather than domain names. The hash table for 4) is built by the program CTRL.SAI. REFORM.SAI currently expects the input to be an IBM-format tape written in EBCDIC with DCB = (LRECL = 135,BLKSIZE = 2700, RECFM = FBA). Other blocking factors and record lengths may be accommodated by altering the procedure READ. The output from REFORM.SAI is in the form of 18 word records. The first two words of each record contain a six character ASCII file name, and the remaining 16 words contain the 80 byte card image.

This output file is sorted on the file names to group the records by file, and we then run the program FILES.SAI to create the data base files from the sorted file. FILES.SAI also creates a file of the names and sizes of these data files for use in building FILES.DEF.