# MANIPULATING DESCRIPTIONS OF PROGRAMS FOR DATABASE ACCESS

P.M.D.Gray and D.S.Moffat

Dept. of Computing Science,

University of Aberdeen, Scotland, U.K..

## ABSTRACT

A method is described for manipulating descriptions of programs to access Codasyl Databases to meet a specification given in relational algebra. The method has been implemented as a Prolog program which is compared with the previous Pascal version. The methodology is discussed as an Automatic Programming technique which explores the transformations on a program induced by changes of data structure representation at two levels.

## I INTRODUCTION

The problem of generating equivalent programs under changes of data representation is an important one. In the case of list processing, a change of data structure representing sets of objects and their relationships can completely change the program. The same applies to Codasyl databases which are essentially enormous list structures on secondary storage. However because of the variety of redundant pointers it is possible to traverse the same list structure in many different ways. Thus it is not just a question of changing the program but of generating alternative programs whose run-times, because of disc access, may differ by factors of 10 or more.

This paper concerns the manipulation of abstract descriptions of such programs. A query is formulated in a functional language (relational algebra) which specifies the logical relationships between the retrieved data values and the stored data items but does not specify the sequence used to access them ( the access path ). The aim is to generate a program that produces the desired items efficiently by exploring a variety of alternative program structures, which are the consequence of following different access paths.

A method of doing this has been developed (Bell 1980) and embodied in a system (ASTRID) (Gray 1982) for typing in queries in relational algebra and generating and running programs on Codasyl databases (IDS-II and IDMS). From the user's point of view the benefits are twofold.

1. It gives the user a relational view of the Codasyl database. Thus he is able to think about his retrieval problem in terms of table manipulations using the high level operations of relational algebra instead of having to work at the low level of record access operations following pointers through the database and embedding these operations in Fortran Code.

2. He can write complicated multi-line queries that compute derived data both from records and groups of records (averages, counts etc.) and appear to generate several intermediate tables. The system will endeavour to find an access path that computes the same result without storing these tables, which could be very costly for large databases. The program generated may be quite complicated to write by hand and should be competitive with a trained programmer's code.

The system goes through several stages. First the user types a query in relational algebra which is parsed and checked. Then it is manipulated at two levels. At the top level the query is rewritten still in algebraic form using rewrite rules so as to assist transformations at the next level. The lower level uses a concrete representation of the Codasyl data structure by a traversal (see below). The system reads in a number of stored traversals for each relation. These have each to be manipulated and combined in various ways to satisfy the requirements of the query. Some combinations will represent very slow and inefficient programs and be discarded. However this cannot be done immediately, as a good program for part of the query may later turn out to be second best after modification to fit the remainder. Finally the descriptions are costed according to information on database access times and the selected version is used to generate Fortran code to run against the actual database. The system is oriented towards complex queries accessing thousands of records which can only run in batch producing substantial printout. Thus it is not the run-time for the translator which matters but the complexity of query which it can handle. Currently other systems only handle a very restricted relational view or a rather restricted query language.

The ASTRID system was originally written in Pascal. More recently the two levels of manipulation have been rewritten in Prolog. This paper describes the basic methodology and shows how Prolog is well adapted to this task.

The layout of the paper is as follows. Section {II} describes some transformations which affect the resultant program but are best carried

out on the relational algebra in Prolog. Section {III} describes the basic notion of a traversal and how it is used to represent a piece of program. Section {IV} describes the combination of traversals and how this is used to build descriptions of more complex programs. Section {V} illustrates some of the Prolog used to combine traversals and discusses its advantages and snags in this application. The final section draws conclusions for future work.

### A. Relation to Other Work

Burstall and Darlington (1977) describe a system for specifying a program by recursion equations. These can be manipulated and play a role similar to relational algebraic expressions in our system. They discuss a way to rewrite the abstract program given a concrete data representation in terms of a "coding function". However our use of a traversal represents the data in a rather different way. Apart from Tarnlund (1978) few have addressed the problem of efficient access to relations using information about the mode of storage. Tarnlund has studied ways to answer queries efficiently by representing them as theorems to be derived in the first order calculus and looking for efficient derivations where relations are held as a binary tree structure.

## II RELATIONAL ALGEBRA TRANSFORMATIONS

The user asks his query in relational algebra. We first describe this and then see how the system improves the query by rewriting it.

### A. Relational Databases

A relation is a set of tuples each containing values for a fixed set of attributes. Viewed as a table the attribute values are in columns. A relational database usually contains several relations which have attributes in common. The examples used come from a database on World Cup football results. The two relations of interest are shown in Table 1.

Table 1. Relational View of World Cup Database

STADIUM_ALLOCATION

| year | group | game | stadium | date |
|---|---|---|---|---|
| 1978 | 1 | 1 | Buenos Aires | 2 Jun |
| 1978 | 1 | 2 | Mar_del_Plata | 2 Jun |
| ... | | ... | | |

GROUP_PLACINGS

| year | group | team | placing |
|---|---|---|---|
| 1978 | 1 | Italy | 1 |
| 1978 | 1 | Argentina | 2 |
| ... | | ... | |

### B. Relational Algebra

Relations can be treated as tables and new relations derived from them by the operations of relational algebra. The operations used are adapted from Codd. They are selection, projection, join, extend and group_by (Gray 1981). The join operation is a generalised intersection, formed from the cartesian product of two relations by selecting those tuples with matching values for the common attributes. A typical query starts by joining several relations, then selects tuples, then extends and or groups these tuples and finally projects to required columns.

The relational algebra can be rewritten, just like standard algebra, by using rewrite rules in PROLOG. We have 17 such rules with special predicates for handling commutation. A typical transformation would move a projection operation(*%*) in an expression involving join(*) and selection(;) to ease the join method.

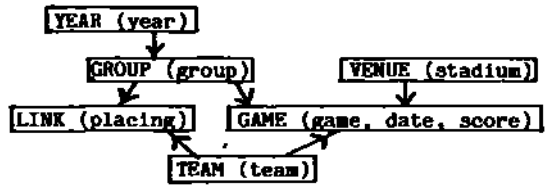
```
(STADIUM_ALLOCATION ; [stadium = "Cordoba''] %year,
 group)*(GROUP_PLACINGS  ;[placing=1]  *year,group)
            becomes
(STADIUM_ALLOCATION  ;[stadium="Cordoba"]  *
 GROUP_PLACINGS  ; [placing = 1 ])  %year,group
```

## III TRAVERSALS of CODASYL DATABASE STRUCTURES

Although the user thinks of relations just as tables, they are actually complicated doubly-linked list structures. At the second level of transformation we need to represent possible paths through these structures by traversals in order to search for an efficient one. Thus we first explain the Codasyl "set" relationship used to link different records. We then see how a number of alternative "base traversals" can be defined for each relation and held on file.

A Codasyl database consists of sets of records of the same type which are linked by pointers to other records in the set and to a common owner record which uniquely identifies an instance of a given set type. Figure 1 shows the linkages between records in the World Cup database.

**Figure 1. Bachmann diagram of World Cup Database**




### A. Traversals

We can now define a traversal of a relation more precisely. It is a description of a piece of code which realises the tuples of the relation one

at a time by accessing the records in some sequence following the set pointers and modifying the values as necessary. Thus it is a generator for a relation. Corresponding to each relation stored in the database ( e . g . GROUP_PLACINGS) we hold on file one or more base traversals. Each one is essentially a description of a piece of code with a number of nested loops.

We have a notation for traversals as follows. Internally it is represented by a Prolog list structure. There are three obvious base traversals of STADIUM-ALLOCATION and two for GROUP_PLACINGS. Each {SA} traversal visits the same number of GAME records, generating one tuple for each.

```
S(YEAR) -> D(GROUP) -> D(GAME) -> U(VENUE)   {SA1}
V(VENUE) -> D(GAME) -> U(GROUP) -> U(YEAR)   {SA2}
B(GROUP) -> U(YEAR) -> D(GAME) -> U(VENUE)   {SA3}
S(YEAR) -> D(GROUP) -> D(LINK) -> U(TEAM)    {GP1}
B(GROUP) -> U(YEAR) -> D(LINK) -> U(TEAM)    {GP2}
```

Here S means a singular set access to visit-all records of a given type (there is only one set owning all year records), D mean go down to visit all member records belonging to the given owner using the appropriate set type (if this is ambiguous it is specified) and U means go up to visit the owner of a given record, V means direct access to the record containing a value (usually given by selection). B means visit every record of that type in the database, In an Algol-like syntax we can represent the corres ponding code for SA1 as : -

```
for each YEAR record do
  for each GROUP record owned by YEAR do
    for each GAME record owned by GROUP do
      for the VENUE owner of GAME do
        print YEAR.year, GROUP.group, GAME.game,
              VENUE.stadium, GAME.date.
```

Thus each arrow in a traversal represents an inner level of nested code. Note that the record generations such as D(GAME) in SA3 must follow those such as B(GROUP), which generates the owner for GAME, but they need not be consecutive.

## IV COMBINATION & MODIFICATION of TRAVERSALS

Corresponding to every algebraic operation on a given relation there is a modification to Its traversal which produces a derived traversal, which is a generator for the new relation. Thus the method is complete. This derived traversal can then be modified by the next operation and so on. For example a selection can be done by inserting "if (year=1978) then" just after "for each YEAR record do". The resulting traversal depends somewhat on the order of application of operations specified by the user. However many of these are commutative and the order of others can be improved by top level rewriting.

### A. Combination by JOIN

Since Join is based on a cartesian product it can be formed by a nested for loop with one iteration for each record type involved. This is very similar to a traversal structure and it turns out that the traversal representing the join can often be formed just by concatenating parts of the separate traversals {Bell 1980, Gray 1981}. The selections for matching are then performed automatically by the factthata Codasyl owner record will in many cases be linked to just those records whose values would have been selected by the join operator! Let us consider examples of this using

RES:= STADIUM-ALLOCATION joined_to GROUP-PLACINGS

If we use SA1 and GP1 then these both have "common start" section.

S(YEAR) -> D(GROUP)

which generates the common attributes in the two cases. If we concatenate the traversals keeping one copy of the common start we get

```
S(YEAR) -> D(GROUP) -> D(GAME) -> U(VENUE) ->
                       D(LINK) -> U(TEAM)
```

we can also get in the other order :-

```
S(YEAR) -> D(GROUP) -> D(LINK) -> U(TEAM) ->
                       D(GAME) -> U(VENUE)
```

Both traversals correspond to nested loop code which will produce the desired tuples though in a different sequence. Which is best depends on subsequent selections. If a selection on "placing = 1" is made after "D(LINK)" then the second method is best as it visits fewer records.

One can also join traversals where the head of one traversal matches the tail or middle of the second. We can do this with the alternative traversals SA2 & GP2 giving :-

```
V(VENUE) -> D(GAME) -> U(GROUP) -> U(YEAR) ->
                       D(LINK) -> U(TEAM)
```

We notice here that a B(GROUP) since it visits all records can match a U(GROUP) which visits only certain records because join has the properties of an intersection.

The second traversal (using SA2,GP2) would be preferred if a subsequent selection were made on stadium as it could use V(VENUE) efficiently. General conditions for choosing an optimum are discussed in (Esslemont & Gray 1982).

## *1* OVERVIEW of the JOIN ALGORITHM in PROLOG

The basic method is given in Figure 2. It starts by reading in a number of traversals for each relation and holds them as unit clauses trav(X). The term X contains a record generation list giving the sequence of record and set accesses, which we have symbolised. The procedure join trav (see below) then picks the first clause

for each relation and tries to find an overlap in accordance with the conditions given in (Bell 1980). Prod overlap is called twice with the record generation lists reversed in order to try the two cases of common start and likewise for head to tail (IV.A). If this is successful the result traversal is asserted. A 'fail' clause then causes backtracking and another pair of traversal clauses is chosen thus trying all combinations of the operand traversals. The 'fail' also has the effect of reclaiming much-needed space once the traversal is safely asserted. If all attempts fail an operation node to join by sort-merge is inserted.

It is possible for a traversal to pass through two instances of the same record type. In order to distinguish which instance is being used for accessing subsequent record types it is necessary to assign a unique number to each record generation element in the traversal. Correspondences are established by clauses of the form equiv_curr(X.Y).

**Figure 2. PROLOG Version of Traversal Join Method**

```
join_trav(Rel1,Rel2,Rel3) :-
  common_columns(Rel1,Rel2,ComCol,NumComCol),
  trav(Rel1,_,_,nds_list(Nds1),recg_list(Rg1)),
  trav(Rel2,_,_,nds_list(Nds2),recg_list(Rg2)),
  exists_nondup_list(ComCol,Nds1,Nds2,Rg1,Rg2),
  (retractall(equiv_curr(_)),
  prod_overlap(Rg1,Rg2,Rg3,Rg4,NumComCol) ;
  retractall(equiv_curr(_)),
  prod_overlap(Rg2,Rg1,Rg3,Rg4,NumComCol) ),
  assert_trav(Rel3,Rg3,Rg4),
  fail.
join_trav(_,_,_).
```

### A. Effect of Joining Modified Traversals

Traversals which have been modified by selection, extension, projection or group-by will have elements in their record generation lists to indicate these operations(operation nodes). Such traversals are joined as before but with all operation nodes being copied directly into the result traversal.

### B. Comparison *of* Pascal and Prolog Versions

The Pascal version takes several thousand lines whereas Prolog needs several hundred and is much easier to read and modify. Pascal is a very much wordier language for list processing. Also one has to write multiple versions of many functions such as "member" because the type of list argument must be known at compile time. Further the use of Prolog Definite Clause Grammars saves pages of recursive Pascal procedures to parse base traversals etc.. Finally because Pascal has no backtracking facilities it has to keep returning sets of alternative combined traversals and currently runs out of list space on large queries. The Prolog version can handle these because it reclaims space following fail.

## VI CONCLUSIONS

Although the direct use of Codasyl databases for storage of facts is unlikely in A.I. the general problem of generating programs that traverse and manipulate list structures is important and the techniques described could have other applications. The methodology used is :-

1. Arrange that the specification of the result to be computed by the generated program is given in functional form such as relational algebra but not in procedural form with loops and assignment. This is easier for the user to think about and also does not commit him to an unsuitable representation. It allows easier overall program transformation; in particular some transformations are easier in the functional form than the traversal form.

2. Prolog is particularly suitable for this work because of its good list-matching and back-tracking facilities. The use of "assert and fail" was necessary, but given this it out-performs Pascal by running larger problems in the PDP 11 address space in similar time.

## ACKNOWLEDGEMENTS

The rewrite rules described in section II were developed by T.N. Scott (now at SCICON, London). Ben du Boulay gave us many valuable comments during the preparation of this paper. The generous assistance of the U.K. SERC is also acknowledged.

## REFERENCES

[1] Bell R., "Automatic Generation of Programs for Retrieving Information from CODASYL Data Bases", PhD Thesis, Aberdeen University,1980.

[2] Burstall R.M. & Darlington J. "A Transformation System for Developing Recursive Programs", JACM, (1977), pp 44-67.

[3] Esslemont P.E. & Gray P.M.D. "The Performance of a Relational Interface to a Codasyl Database" in Proc. BNC0D-2, ed. S.M.Deen and P.H. Hammersley, Bristol 1982.

[4] Gray, P.M.D. "The GR0UP_BY Operation in Relational Algebra", in "Databases (Proc. BNC0D-2)" ed. S.M. Deen & P. Hammersley (1981),pp. 84-98.

[5] Gray, P.M.D. "Use of Automatic Programming and Simulation to Facilitate Operations on Codasyl Databases" in "State of the Art Report DATABASE", Series 9 No.8, ed. M.P.Atkinson, Pergamon Infotech (Jan 1982),pp 346-369.

[6] Tarnlund S-A, "An Axiomatic Data Base Theory" in "Logic and Data Bases", ed. Gallaire & Minker (1978), pp. 259-289.