EFFICIENT PROLOG ACCESS TO CODASYL AND FDM DATABASES

P.M.D. Gray

Dept of Computing Science*
University of Aberdeen,
King's College,
ABERDEEN AB9 2UB.
Scotland . U.K.

ABSTRACT

A method is proposed for accessing Codasyl and Functional Data Model databases from Prolog. It uses tokens or references to objects as values, instead of the more usual method of using relational attributes as values. It is shown how to implement an n-ary relational view based on such tokens, and how this view relates to code used with Codasyl and FDM databases. A method for improving the efficiency of joins in this formalism is given, based on that implemented and tested in the ASTRID relational algebra system. Directions for future work are given, including suggestions on storage of more general AI structures.

1. INTRODUCTION

Many authors have suggested that unit clauses in Prolog should be represented as n-ary relations and stored directly in a relational database. In particular, Lloyd and Naish (1981) have implemented such a system, using a hashing scheme which allows partial-match retrieval on a variety of combinations of attribute values. This paper explores an alternative method which allows access to a wider range of databases than purely relational ones. It represents n-ary relations by Prolog clauses that are written in terms of predicates referencing surrogate keys.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery To copy otherwise, or to republish, requires a fee and/or specific permission

© 1985 ACM 0-89791-160-1/85/005/0437 $00 75

These keys may take the form of database keys for records in a Codasyl database, or of references to entities in a Functional Data Model database (Shipman 1979).

We suggest the use of n-ary predicates in Prolog because it preserves an n-ary relational view of the database, which is convenient for many manipulations. In particular, it allows one to formulate queries and updates in a language such as Query-by-Example (Zloof 1977), and to reason about them, check integrity constraints, and help the user by filling in missing links, all in Prolog, as done by Neves et al (1983). This can all be done at the n-ary level without considering how the view is implemented at the lower level.

*Visiting Professor at
Dept. of Computer Science,
Univ. of Western Ontario,
LONDON, CANADA N6A 5B7

2. GENERATING N-ARY RELATIONS THROUGH EVALUABLE PREDICATES

Let us consider how to write Prolog clauses that return tuples from n-ary relations, stored in a Codasyl database. Consider the Bachman Diagram of Figure 1.

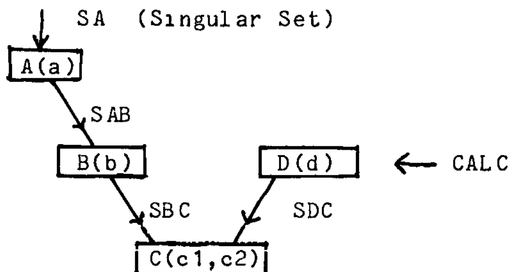

Figure 1 - Codasyl schema for relation R(a,b,c1,c2,d)

Suppose that we have a relation R(a,b,c1,c2,d), whose primary key is <a,b,c1>. In the schema of figure 1 there will be one occurrence of record type C for every tuple in R. The values of attributes c1,c2 are given by the corresponding fields c1,c2 of the C record, whilst the attribute values for a,b and d are taken from the fields stored in the instances of owner record types A,B and D respectively. In an actual database these records might own other record types as well, which are not shown. All instances of type A are owned by the singular set SA. The other set types used are shown on the diagram as SAB, SBC and SDC. The Codasyl-relational mapping used is that used in the ASTRID system (Gray & Bell 1979), which is very similar to that proposed by Zaniolo (1979).

We can write a Prolog definition for R as follows -

```
(1) R(A,B,C1,C2,D) -
    SA(A#), GetAa(A#,A),
    SAB(A#,B#), GetBb(B#,B),
    SBC(B#,C#), GetC1(C#,C1),GetC2(C#,C2),
    SDC(D#,C#), GetDd(D#,D).
```

This works as follows. Every Prolog variable name that ends in '#' is to be instantiated with a value which is a database key, giving direct access to a particular record instance (i.e. a form of pointer to a record on disk). The predicate SA is an 'evaluable predicate' which returns each possible value for its argument A# in succession, taken from the singular set SA. The predicate SAB, if given the value of A#, will return references to successive records from the set SAB owned by A#, and so on. The predicate is also invertible, so that if given a value for B# it will return the owner value in parameter A#. If given values for both A# and B# it will check that B# is owned by A# via the set SAB, and succeed or fail accordingly. This follows the usual conventions for Prolog predicates.

The predicates SAB, SBC and SDC are behaving very much like a binary relation containing the instances of a Codasyl set, as suggested by Rosenthal and Reiner (1982). However, by considering them as Prolog predicates, we do not have to exhibit an explicit table of values of database keys. The keys can only be used to instantiate the predicate at run-time and, by convention, they would not be printed or stored, in keeping with Codasyl practice. It would, of course, be very easy to support the predicate SAB, since the instruction FIND (FIRST, SET=SAB), followed by FIND (NEXT, SET=SAB) would load the successive values of B# into the currency register for record type B, given the value of A# as current of record type A. Similarly, given the value of B#, the instruction FIND (OWNER, SET=SAB) would load the value of A# into the current of run unit.

The predicate GetAa is also an evaluable predicate. It is used to model the GET verb of Codasyl. It is given the database key A# for a record of type A and returns the value for attribute 'a'. The predicates GetBb, GetC1, GetC2 and GetDd are defined similarly to return the attributes their names suggest (there is a predicate for every field of each record). Whether the predicates are invertible depends on the implementation. We shall assume that record type D is stored by hashing (CALC mode) on attribute 'd', and thus given the value of 'd', GetDd(D#,d) will return a value for D#. In the case where values are given for both D# and 'd' the predicate will check the value retrieved for 'd' against that given, and succeed or fail accordingly. In this case the predicate can act as a selection, as can all the other 'Get' predicates. In terms of the Entity-Relationship model GetAa(A#,A) is an entity relation, whilst SAB(A#,B#) is a relationship relation.

We shall not discuss matters of

implementation further except to remark that the techniques used by Buneman et al (1981) to implement the FQL interface to the SEED Codasyl DBMS by passing currency register values should be easily adaptable to provide a Prolog implementation. In our case, we (Gray, Moffat & du Boulay 1984) are constructing an interface from Prolog to a Persistent Heap database (Atkinson et al 1981) which has been used to implement Shipman's DAPLEX language (Atkinson & Kulkarni 1983, Fox et al 1984). Our version of Prolog will be able to call on evaluable predicates whose body is written in the implementation language PS-Algol, and thus can return references to DAPLEX entities. We can treat a DAPLEX database very like a Codasyl database, (Gray 1984b), and thus use the same techniques in Prolog to represent relations as those described above. Let us now consider the consequences of using this technique, and how we can optimise access to either kind of database from Prolog.

3. GENERATING RELATIONS BY NESTED LOOPS

In order to understand how our Prolog definition for R given above can be used to generate the tuples of an n-ary relation, let us consider a piece of code written in a DAPLEX style that would visit the records of the database and print out the tuple values.

```
For each A# in SA do
  For each B# owned by A# in SAB do
    For each C# owned by B# in SBC do
      For the D# owner of C# in SDC do
        Print A#.a, B#.b, C#.c1, C#.c2,D#.d
```

Here we use a postfix notation for record field selectors. The indentation denotes nesting of loops. The fact that we are descending the hierarchy of the key <a,b,c1> ensures that each instance of record type C will be visited once only, and thus all the tuples of the relation are generated once only (we are ignoring problems of null values).

Let us now consider the Prolog query.

```
?- R(A,B,C1,C2,D),
     write([A,B,C1,C2,D]), nl, fail.
```

This forces repeated backtracking through the definition of R and will print the tuples of R in the same sequence as the nested loop definitions. In fact we can consider the nested loops as a compiled description of the way in which a standard Prolog interpreter would evaluate the definition of R when generating values, because it works depth first and left to right.

4. CHOICE OF ALTERNATIVE ACCESS PATHS

We can write down at least two other definitions of R in Prolog, which will generate the same tuples but in a different sequence. All three definitions can also be used, of course, to test for the existence of a tuple with a particular set of values for A,B,C1,C2 and D. The definitions are:

```
(2) R(A,B,C1,C2,D) -
    GetDd(D#,D), SDC(D#,C#), GetC1(C#,C1),
    GetC2(C#,C2), SBC(B#,C#), GetBb(B#,B),
    SAB(A#,B#), GetAa(A#,A).

(3) R(A,B,C1,C2,D):-
    RB(B#), GetBb(B#,B),
    SAB(A#,B#), GetAa(A#,A),
    SBC(B#,C#),GetC1(C#,C1), GetC2(C#,C2),
    SDC(D#,C#), GetDd(D#,D).
```

The three definitions differ in the order of the goals, and, in particular, in the first goal to be attempted. This goal represents the initial mode of database access. For definition (1) it is via the singular set SA. For definition (2) it is via GetDd(D#,D), which is particularly good if D is known, as it can use hashing for fast selective access. For definition (3) the evaluable predicate RB(B#) is used to find all occurrences of record type B in the database, searching pages in turn. This is slow, but it may be advantageous if many occurrences of record type B occur, round which occurrences of record types A & C are clustered, since it saves revisiting pages.

It is well known that differences in the initial mode of access can make a big difference to the speed of finding records in a Codasyl database. Indeed, the physical design of Codasyl databases is largely concerned with providing alternative access paths to records, and clustering associated items together. Thus we are interested in making good use of these facilities from Prolog.

Interestingly, Warren (1981b) discovered that differences in the order of goals, and in the initial access, caused considerable differences in the speed of Prolog programs accessing a geographic database. His database just used the normal internal storage of Prolog unit clauses in memory, which

corresponds to a very simple relational storage scheme, based on hashing of only one argument. His paper presents a method for re-ordering the goals to take account of the known average selectivity of the predicates, (which corresponds to the average cardinality of the corresponding Codasyl sets). He also tried to use selection information early to reduce the generation of tuples, as when we use a selection on the value of 'd' in combination with definition (2) of R to restrict the numbers of records visited. Warren pointed out that his method was very similar to that used by Selinger et al (1979) in the optimiser for System R. An experimental study by Esslemont and Gray (1982), has shown that a similar method can be used for Codasyl databases, except that the relative weightings of the different access methods are different, and it is not so easy to make accurate estimates of cardinalities and page changes.

5. IMPLEMENTING JOIN OPERATIONS

Many of the operations we want to do in Prolog correspond to the selection and join operations of relational algebra. Following Warren's methodology, we shall see how a Prolog program could inspect a Prolog query formed against definitions of relations as given earlier, and how it could make significant optimisations, using techniques already established for relational queries on Codasyl databases by Bell(1980), Gray(1984b).

Consider the following piece of Prolog .-

```
RES(A,B,C,D,E,G)  .-
R(A,B,C,_,D), R1(A,B,E,G).

R1(A,B,E,G)   -
GetGg(G#,G), SGE(G#,E#), SBE(B#,E#),
GetBb(B#,B), SAB(A#,B#), GetAa(A#,A),
GetEe(E#,E).
```

Here the relation RES is the join of the two relations R and R1 on the common columns A and B. Let us consider definition 3 for relation R. If we just use the Prolog interpreter simplistically then we shall get a very inefficient evaluation of RES, since the generator RB(B#) will be sequentially enumerating the database, only to fail many times inside R1. However we can combine the definitions together to give a much better definition for RES thus -

```
RES(A,B,C1,D,E,G)  :-
    GetGg(G#,G), SGE(G#,E#),
    SBE(B#,E#), GetBb(B#,B),
    SAB(A#,B#), GetAa(A#,A),
    SBC(B#,C#), SDC(D#,C#),
    GetEe(E#,E), GetC1(C#,C1), GetDd(D#,D).
```

This definition is particularly effective if it is used in a context where the value of G is known. The leading GetGg can then be used for hashing to find the value of G#. We have eliminated the undesirable predicate RB(B#), since the value of B# is known from SBE(B#,E#), and there is no point in using RB to test B#, since we know from the schema that it is a reference to the right type of record.

The new definition of RES has been obtained quite simply from the originals by taking the union of the goals in the two clauses, and preserving the relative ordering of the goals within each clause. This is satisfactory since we are just taking a conjunction of goals. However, it is not always as simple as this.

The general method is derived from that given in Gray(1981,1984a) based on the thesis of Bell(1980). This considers the problem of joining two "traversals" of Codasyl sets, which are nested loops of code, like those discussed earlier, acting as generators for n-ary relations. The thesis gives the conditions under which the result of the join can be represented by a sequence of more deeply nested loops. If these conditions are not satisified, then the join must include explicit operations for the removal of duplicates and matching of values. In the ASTRID system (Gray & Bell 1979, Gray & Moffat 1983), which implements this technique for a general relational algebra on IDMS and IDS-II databases, the join is done by sorting and merging, where it cannot use Bell's method.

In the published papers we describe the nested loops by a notation with a sequence of arrows. Thus the definitions of R, R1, and the result RES, would be written as -

```
R.  B(B) -> U(A SAB) -> D(C:SBC) -> U(D·SDC)
R1: V(G) -> D(E SGE) -> U(B:SBE) -> U(A:SAB)
RES V(G) -> D(E SGE) -> U(B:SBE) ->
           U(A.SAB) -> D(C·SBC) -> U(D:SDC)
```

Here the mode of generation is given by B, V, S, U or D, meaning Basic pagewise scan, Value hashing, follow Singular set, go Up to owner or go Down to

member(s) respectively. The combination of traversals is seen as a combination of two linear sequences which contain a common segment (generating the common columns). In this case the common segments are B(B) -> U(A·SAB) and U(B.SBE) -> U(A:SAB). The generator B(B) is considered to match U(B·SBE) as it is more general.

6. CONDITIONS FOR JOIN BY TAKING CLAUSE UNION

The conditions are as follows (taken from Bell(1980), Gray(1984a))·-

Condition (i)

"The common column values must be derived from the same fields of the same instances of the same record types, accessed via the same sequence of set types in the two cases (but an initial B access matches any other generation of the same record type)"

In the Prolog case we can check this by pattern matching e.g.

```
SAB(A#,B#), GetBb(B#,B), SBC(B#,C#),
                         GetC1(C#,C1)
```

with

```
SAB(A#,X#), GetBb(X#,B), SBC(X#,Y#),
                         GetC1(Y#,C1)
```

Most of the checking is done by matching the predicate symbols SAB, GetBb, etc., since this checks the record names, field names and set types. The parameters may have different names in the two cases (e.g. B# and X#). The matcher must also be careful when dealing with access to two different instances of the same record type, as below where B# is not necessarily the same as B1#

```
SBC(B#,C#),... SBC(B1#,E#),...
```

Condition (ii)

"The common column generation part must be the leading part of one definition".

This ensures that accesses to other records are determined by the instances of records containing the common columns, and not vice versa. In the Prolog case it may just require re-ordering of the definition, using the fact that many of the predicates are invertible. In fact this is what we have done in writing down the different definitions of R.

Condition (iii)

"The two traversals may contain selection predicates, provided these are all copied into the result so as to preserve their relative position"

Thus, for example, if R1 had included extra selections like "..,Gt(A,7),.." to test that A exceeds 7, and R had included "..,C='Aberdeen',..", then both these goals must be included in the union of the goals from the two clauses· "..,Gt(A,7),...C='Aberdeen',..". This is allowable since a join behaves like a generalised intersection, and the Prolog form tests the conjunction of the goals, which must both be satisfied. We choose the order of the selection goals so that they take place as early as possible after the values which they test become available.

Condition (iv)

"The common column values must be generated without duplication; that is, the combination of their values must uniquely determine the instances of the records containing them"

Note that this does not mean that we can only do one-one joins, since the same record instances which produce a single value for the common column attributes can own sets which produce multiple values for the non-common attributes on both sides, which produces a many-many join result.

To justify this condition consider the example above, where we had two sequences.

```
...SBE(B#,E#), GetBb(B#,B), SAB(A#,B#),
                            GetAa(A#,A),...
...RB(B#), GetBb(B#,B), SAB(A#,B#),
                            GetAa(A#,A),...
```

In the combined definition of RES we eliminated the second sequence, as being equal to the first and redundant. However, the join is defined just by the equality of the common column values A=A, B=B, but in order to make the sequences equal we have assumed A#=A#, B#=B# also. Thus the method only works where equality of the fields generating the common columns implies equality of the record instances.

This condition is commonly satisfied by a Codasyl database where a sequence of records forms a hierarchy.

Thus suppose in the given example that "a" is a key for record A#, and that "b" is a set key for set SAB. Thus no two type A records can have the same value of "a", and no two type B records in set SAB with the same owner record A# can have the same value of "b". Thus <a,b> functionally determines A# and B#. It does not matter whether we ascend the hierarchy from B to A in our access path, or whether we descend it.

Typically, Codasyl databases include a number of alternative hierarchies of owners for a common record type such as C in figure 1. When we come to join the corresponding relations we often find that the common columns are record keys or set keys for one of the hierarchies, so that the set of tuples with given common column values is found by following the Codasyl set pointers. In some sense the set pointers behave as "precomputed" join links, and we can regard many Codasyl databases as relational databases where the match operation for the join has been precomputed by joining the relevant records together in a chain!

7. FURTHER WORK

This study suggests a number of interesting directions for future work. Firstly, there is the question of the generation of alternative Prolog descriptions of relations. In the ASTRID system, alternative definitions are stored on a file, allowing the database administrator to make use of special peculiarities of the database, for example some sets may be used to represent repeating groups, or some items to represent values implicit in the order of a set (as explained in Gray 1984b). However, in the Prolog system it may be possible to reason about and generate these alternative definitions, instead of storing them.

Secondly, there is the question of optimisation of other sorts of join, such as that based on the value of an attribute (for example "d") which is computed for one relation and used to give direct access by hashing to the record generating that value in the other relation. This should be fairly easy to implement, just as in the ASTRID sytem.

Thirdly, there is the question of optimising operations such as group-by, and set difference. These operations on sets must be expressed in Prolog by use of the special "setof" meta-predicate, proposed by Warren (1981a). Some of the optimising techniques used in ASTRID to simplify differences and joins onto the results of a difference or group_by (Gray 1981,1984b) should be applicable in the Prolog formulation.

Lastly, and most importantly, there is the question of providing clean interfaces between Prolog and a Functional Data Model or Codasyl database. There is clearly a role for type information, and investigations are under way to use an Abstract Data Type definition as the basis for the interface. Here, checks are made on the types of arguments such as A#. These appear to be atomic to Prolog, but details of their representation and of the corresponding implementation of evaluable predicates such as SAB and GetAa are hidden from Prolog, as described in Gray (1984c).

8. CONCLUSION

We have studied how to use Prolog to get access to information stored in Codasyl or Functional Data Model databases. We have used a relational mapping to n-ary relations at the top level, but at a lower level we have used Prolog evaluable predicates which manipulate tokens which reference objects stored in the database. This is distinct from the usual proposals which store the n-ary relations directly. In this paper we have considered how to use information about the tokens to implement in Prolog an optimiser for making efficient use of the Codasyl sets and storage techniques. We have been able to make use of many of the techniques of an existing relational algebra query optimiser, from the ASTRID system, and the Prolog formulation has thrown a new light on the method used for joins in that system, besides being useful in Prolog. In the longer term we envisage the use of such tokens to point not just to simple records, but to more general objects, as in object-oriented programming, and it opens up the possibility of a wider range of database use from Prolog than the straightforward use of relational databases.

9. ACKNOWLEDGEMENTS

The author is grateful to David Moffat for many helpful comments. This work is supported by a grant from the U.K. Science and Engineering Research Council.

10. REFERENCES

[1] Atkinson, M.P., Chisholm, K.J., and Cockshott, W.P., (1981), "PS-Algol. an Algol with a Persistent Heap", ACM SIGPLAN Notices 17 (7).

[2] Atkinson,M.P. & Kulkarni,K.G. (1983), "Experimenting with the Functional Data Model", in "Databases: Role and Structure", P.M.Stocker (ed.), Cambridge Univ. Press.

[3] Bell,R. (1980), "Automatic Generation of Programs for Retrieving Information from CODASYL Data Bases", Ph.D Thesis, University of Aberdeen, Scotland.

[4] Buneman,O.P., Menten,L. & Root,D. (1981), "A Codasyl Interface for Pascal and Ada", Research report, Dept. of Computer Science, Univ. of Pennsylvania, Pa.

[5] Esslemont,P.E. & Gray,P.M.D. (1982), "The Performance of a Relational Interface to a Codasyl Database", in Proc. 2nd British National Conf. on Databases (Bristol), to be published in the Computer Journal.

[6] Fox,S., Landers,T., Ries,D.R. & Rosenberg,R.L. (1984), "Daplex Users Manual", Report CCA-84-01, Computer Corporation of America, Cambridge, Mass.

[7] Gray,P.M.D. & Bell,R. (1979), "Use of simulators to help the inexpert in automatic program generation", Proc. Euro-IFIP79 Conf., P.A.Samet (ed.), North-Holland, 613-620.

[8] Gray,P.M.D. & Moffat,D.S. (1983), "Manipulating Descriptions of Programs for Database Access", Proc. Eighth International Joint Conference on Artificial Intelligence, IJCAI-83 (Karlsruhe), A.Bundy (ed.), 21-24.

[9] Gray,P.M.D. (1981), "The GROUP_BY Operation in Relational Algebra", Proc. BNCOD-1 Conf., S.M.Deen & P.H.Hammersley (eds.), Pentech Press, 84-98.

[10] Gray,P.M.D. (1984a), "Implementing the Join Operation on Codasyl DBMS", in "Databases: Role and Structure", P.M.Stocker (ed.), Cambridge Univ. Press.

[11] Gray, P.M.D. (1984b), "Logic, Algebra and Databases", Ellis Horwood (Wiley), Chichester.

[12] Gray, P.M.D. (1984c), "The importance of Knowledge representation in Inference Techniques", Proc. Alvey Inference Workshop, Sept.1984, ed. R.A.Kowalski, Imperial College, London.

[13] Gray, P.M.D., Moffat, D.S. and du Boulay, J.B.H. (1984), "A Secondary Storage Manager for Prolog", Proc. 1st Alvey workshop on Architectures for Large Knowledge Bases, ed S. Lavington, Computer Science Dept., Manchester, England.

[14] Lloyd,J.W. (1981), "Implementing clause indexing in Deductive Database Systems", Research Report 81/4, Computer Science Dept., Univ. Melbourne, Australia.

[15] Neves,J.C., Anderson,S.O. & Williams, M.H. (1983), "A Prolog implementation of Query-by-Example", Proc. 7th International Computing Symposium (Nurnberg).

[16] Rosenthal,A. & Reiner,D. (1982), "Querying Relational Views of Networks", Proc. IEEE COMP SAC'82 (Chicago).

[17] Selinger,P.G., Astrahan,M.M., Chamberlin, D.D., Lorie,R.A. & Price,T.G. (1979), "Access Path Selection in a Relational Database Management system", Proc. SIGMOD79 Conf, Boston, 23-34.

[18] Shipman,D. (1979), "The Functional Data Model and the Data Language DAPLEX", SIGMOD 79 Conf, revised version, ACM TODS,6, 140-173.

[19] Warren,D.H.D. (1981a), "Higher-order extensions to Prolog - are they needed?", in "Machine Intelligence 10", D.Michie, J.Hayes, Y.H.Pao (eds.), Ellis Horwood.

[20] Warren,D.H.D. (1981b), "Efficient Processing of Interactive Relational Database Queries expressed in Logic", Proc. 7th VLDB conference, Cannes, 272-281.

[21] Zaniolo,C. (1979), "Design of Relational Views over Network Schemas", Proc ACM SIGMOD Conf (Boston), 179-190.

[22] Zloof,M.M. (1977), "Query-by-Example: A Data Base Language", IBM Sys J. (16), 324-343.