Optimising and Executing DAPLEX Queries using Prolog

N.W. Paton* AND P.M.D. Gray
Department of Computing Science, University of Aberdeen, King's College, Aberdeen AB9 2UB

In this paper a query optimiser for use with functional data model databases is described. The system is in use with a large database of protein structures from which examples are taken. The optimiser, which evaluates alternative paths through the object base, is written in Prolog and integrated with a parser for DAPLEX. The constructs of DAPLEX are easily expressed in Prolog, which has also proved suitable for implementing both the parser and rewrite rules. The implementation of the optimiser using rewrite rules is both concise and extensible.

Received September 1989, revised March 1990

1. INTRODUCTION

One of the characteristics of the relational model which has contributed to its widespread acceptance is the ease with which ad hoc queries can be expressed using declarative query languages. The importance of optimising the declarative queries submitted for execution by non-expert users was quickly recognised, and transformations were devised which took into account the internal representation of the database.\textsuperscript{5,7,15,28,24} This paper is concerned with query optimisation in a functional data model (FDM) database using the query language DAPLEX.\textsuperscript{29} DAPLEX is a language based upon navigation in which complex queries can be written which do not suit the set-based style of SQL. It is of interest because it provides many of the features of a conventional programming language, while still making it possible to use global query optimisation techniques. It is also very suitable for use with an object database. For this purpose, the functions of DAPLEX have been re-interpreted as methods which act on objects stored in the modules of Perlog, a Prolog with modules and persistence.\textsuperscript{12,18}

A number of other query languages including EFDM, an extended version of DAPLEX,\textsuperscript{17} OSQL, a version of SQL\textsuperscript{8} and the functional language FDL\textsuperscript{23} have been implemented for use with functional and object-oriented databases, but the question of how to perform global query optimisation in languages such as these remains largely unaddressed.

In this paper we describe an optimiser for DAPLEX which has been implemented using rewrite rules expressed in Prolog for use with P/FDM,\textsuperscript{12} an integration of Prolog with a functional data model (FDM) database. Note that Prolog is used in P/FDM as an implementation language for a functional database rather than as a logic programming system.

2. OPTIMISATION STRATEGY

P/FDM is being used with a database of protein structure data.\textsuperscript{14,19} The schema diagram of the protein structure database used for examples in this paper is Fig. 1. At the time of writing this database contains some 40 Mb of three-dimensional protein structure data from over eighty proteins. It is important for protein chemists to be able
to express queries in a way which is natural to them, and an optimiser is used to convert queries into efficient Prolog for execution. This has proved to be essential, because a literal translation of many queries would take hours to run instead of minutes.

The general philosophy of the optimiser is to perform query improvement using heuristics, rather than striving for exhaustive optimisation.\textsuperscript{6} Hall\textsuperscript{15} calls this 'amelioration'. Although the optimiser uses a simple cost model, it is effective in avoiding inefficient strategies, and often selects one of the most effective approaches.

THE COMPUTER JOURNAL, VOL. 33, NO. 6, 1990 547

Figure 1. Protein structure database: schema diagram.
The underlying object store uses extensible hashing to relate the object identifier of an object to the object's properties. As explained below, these properties (which are found by applying the appropriate functions) may themselves be object identifiers or lists of object identifiers. These properties are retrieved in a single disc access, and a sequence of such accesses form a navigation path. The present P/FDM back-end performs no caching, and has no control over the clustering of objects on disc. Thus for optimisation purposes it is simply necessary to minimise the number of function calls made. The calculation which estimates the number of function calls made is based upon the cardinality of each class and the average cardinality of each multi-valued function. These cardinalities are obtained by periodically running Prolog programs over the database, and are stored with the class and function descriptors.

In contrast with most relational optimisers, we are concerned with optimising and reordering a path, and are not concerned with set-based optimisations like sort-merge, or with batch operations with take advantage of record clustering. It is likely that this style of optimisation will be common in future object-oriented databases, since they are likely to be paged into large physical memories, which further reduce the relative advantages of set-based operations.

### 2.1 Queries as ZF expressions

A query against an object-oriented database identifies a set of objects which satisfy a number of constraints. For example, the query print the sequence of all three-helices which are five residues long can be expressed in DAPLEX as:

```prolog
for each c in chain
  for each t in threeten such that structure_chain(t) = c and length(t) = 5
    for each i in {start(t) to end(t)}
      print component_name(c), structure_name(t),
      name:absolutepos(c,i));
```

A sample of the output is:

B gl ala
B gl ser
B gl gln
...

Note that the variables `c` and `t` are bound to successive object identifiers taken from sets in the database. Some of these sets are obtained by evaluating functions applicable to objects retrieved from the database. The `such that` clauses restrict the members of a set using a boolean expression which can contain functions applied to variables from enclosing loops. Note that `functions` in DAPLEX can be used to retrieve the attributes of an object, follow relationships between objects, and perform calculations. This style of query navigates through a network of objects following pointers. Although the query looks like nested loops from an imperative language, it can be expressed declaratively in the ZF (Zermelo–Fraenkel) set notation\(^{29}\) as shown below, using the same variable names.

```prolog
{ c.component_name, t.structure_name, r.name | t <- threeten, tc <- structure_chain(t), tc = c, len <- length(t), len = 5, start <- start(t), end <- end(t), i <- [start...end], r <- absolutepos(c,i)}
```

Read declaratively, this says that the answer is the set of tuples formed from the `component_name` of `c`, the `structure_name` of `t` and the `name` of `r` such that `c` is an instance of the class `chain`, `t` is an instance of the class `three ten`, `c` is the `structure_chain` of `t`, the `length` of `t` is 5, `i` is an integer between the `start` and the `end` of `t`, and `r` is the residue at the `i`th position in `c`. The operations which take place are identified rather better by the ZF notation than in DAPLEX. The statements in the body of the ZF expression which contain the operator `<-` are `generators` which return object identifiers or scalars as successive members from a set, while tests such as `len = 5` serve to `restrict` the set of results. The `to` operator is an extension to standard DAPLEX which permits iteration over a range of integers. It is particularly useful when expressing sequence queries in the protein structure database.

If the query is executed as expressed above, in what is the direct mapping from DAPLEX to ZF notation, then all `chain/threeten` pairs will be generated before any `constraints` are applied. The ZF form is a conjunction of goals which can be reordered, provided that the operands of an operator are known before the operator is evaluated, and that the arguments of a `generator` are known before it is called. A more efficient approach is to re-order the ZF-expression to start with entities of class `three ten` – which results in early application of the constraint on the `length` of the helix – and to use `structure_chain` as the sole generator of `chains`.

```prolog
{ c.component_name, t.structure_name, r.name | t <- threeten, len <- length(t), len = 5, c <- structure_chain(t),
  start <- start(t), end <- end(t),
  i <- [start...end], r <- absolutepos(c,i) }
```

The corresponding DAPLEX query is:

```prolog
for each t in threeten such that length(t) = 5
  for each c in structure_chain(t)
    for each i in {start(t) to end(t)}
      print component_name(c), structure_name(t),
      name:absolutepos(c,i));
```

The strategy employed by the optimiser is to reorder the calls in the body of a query, taking into account inverses, constraints and the cardinality of relationships. The more complex structures of DAPLEX such as set operations and aggregate functions are also handled by the optimiser, as described in Section 3.4.

The notion of reordering goals in the body of an expression is a form of conjunctive query optimisation,\(^{26}\)

![Figure 2. Stages in query optimisation (all written in Prolog).](https://example.com/image.png)

\(^{26}\) N. W. Paton and P. M. D. Gray
and has been used by Warren\textsuperscript{11} to optimise relational database queries expressed in Prolog. There are also similarities with the ASTRID system,\textsuperscript{8,12} which maps queries expressed in relational algebra into programs which retrieve data from a Codasyl database. In ASTRID, operations which traverse the set hierarchy of the Codasyl model are reordered to apply strong constraints early and to take account of any clustering of data on secondary storage. However, we are not aware of any use of this technique with functional or object-oriented databases.

A query optimiser for executing a subset of DAPLEX against a Codasyl back-end as part of the Distributed Dbms MULTIBASE is described by Dayal.\textsuperscript{4,5} A class of queries known as tree queries are identified which are suitable for execution by a Codasyl DBMS, and an approach to evaluating different execution strategies is described. While the optimiser described here performs a number of optimisations which are similar to those performed in MULTIBASE, the rule-based implementation strategy and the nature of the code generated in our system are completely different. The architecture of P/FDM has permitted the optimiser to be written as an application of the P/FDM back-end, rather than as part of the kernel of a large DBMS. Using this approach, the parser and optimiser for the full DAPLEX language were written from scratch in only 4 months. The resulting system is of moderate size and is reasonably quick.

3. IMPLEMENTATION

3.1 Overview

In this section a Prolog implementation of an optimiser for DAPLEX is described. The output from the optimiser is a piece of code in the form of a sequence of Prolog goals for execution using P/FDM. The goal predicates, which are generic operations on objects include the following:\textsuperscript{12}

\begin{verbatim}
getentity( <class, >, <instance > )
getentity( <class >, <key >, <instance > )
getfnval( <function >, <args >, <res > )
addfnval( <function >, <args >, <val > )
\end{verbatim}

The predicate getentity/2, given an entity class, returns the instances of that class that are at a time by backtracking. The predicate getentity/3, given an entity class and the (possibly compound) key of an instance of that class, returns that instance by direct access. The predicate getfnval/3, given the name of a function and a list of arguments returns the result(s) of that function at a time by backtracking. The predicate addfnval/3, given the name of a function, a list of arguments and a value of the result type adds that value to the extension of the function. Thus the following piece of Prolog is equivalent to the DAPLEX query in Section 2.

\begin{verbatim}
( getentity( threeten, T ),
  getfnval( length, [T], 5 ),
  getfnval( structure_name, [T], TName ),
  getfnval( structure_chain, [T], C ),
  getfnval( component_name, [C], CName ),
  getfnval( end, [T], End ),
  getfnval( start, [T], Start ),
  int_set( Start, End, I ),
  getfnval( absolutepos, [C, I], R ),
  getfnval( name, [R], RName ),
  write( [CName, TName, RName] ), nl, fail
; true
).
\end{verbatim}

The generated Prolog corresponds to the rewritten ZF expression of Section 2, except for the evaluation of result variables (CName, TName and RName). While it can look rather long-winded, it is intended that the user will be shielded from the Prolog representation of the query by writing in a language such as DAPLEX. The predicate getfnval/3 is similar to a message sent in a object-oriented programming system, in that the implementation of each function is hidden within a Prolog module.\textsuperscript{13} The stages in the transformation of a DAPLEX query into an optimised program expressed in Prolog are given in Fig. 2.

3.2 Parsing

A parser and translator for DAPLEX has been written in Prolog. The parser generates P/FDM notation, which is a list of the elements in a ZF expression, not a relational algebra tree as used by Derrett and Shan\textsuperscript{4} and others. The term structures which make up the P/FDM form of two examples queries are given in Fig. 3. The components of the P/FDM form reflect the facilities offered by DAPLEX. The getentity/3 terms represent the edges of a query graph which has an entity variable at each node.\textsuperscript{14} It is felt that the same internal form could be used as output from query languages other than DAPLEX, such as OSQL.

The parser for DAPLEX makes use of the grammar rule mechanism of Prolog,\textsuperscript{13} in which embedded actions are used for semantic analysis and code generation. When parsing, the variables used in each expression are recorded to give a dependency graph used later in optimisation.

3.3 Preparations For Rewriting

Before the P/FDM form produced by the parser can be rewritten, it is necessary to group related database calls, and to identify relationships between entity variables. The groupings which take place associate each getentity term which acts as a generator with its corresponding getfnvals, and clusters these groupings around relationships. An example of this grouping is given in Fig. 4. The grouped P/FDM form (gpf) supports the reordering of the loops in DAPLEX and facilitates early application of restrictions.

3.4 Rewriting for optimisation

The use of rewrite rules for optimising expressions in relational algebra is well understood.\textsuperscript{9,10} Rewrite rules are a mechanism for replacing one representation of a structure with some alternative equivalent representation. The Prolog rewrite rules estimate the cost of all legal permutations of the elements in the gpf of the query. Examples of rewrite rules expressed in Prolog are given in the Appendix. The advantages of rule-based optimisers are well known\textsuperscript{6} – they are concise, easy to maintain, extensible and safe (rules do nothing unsafe to constructs which they do not know how to optimise).

THE COMPUTER JOURNAL, VOL. 33, NO. 6, 1990 549
Examples:

1. for each p in protein such that source(p) = "human" and resolution(p) < 2.5
   print protein_code(p);
   getentity(protein,P),
   boolean_expr(and,[P],
   getfnval(source,[P],"human"),[],
   getfnval(resolution,[P],Var1),Var1 < 2.5),
   getfnval(protein_code,[P],Var2))

2. for each c in chain such that
   at least 2 s in structure_chain_inv(c) have length(s) = 4
   print component_name(c);
   getentity(chain,C),
   restrict_subquery(structure,S,least(S,2),
   getfnval(structure_chain_inv,[C],S),
   getfnval(length,[S,4]),
   getfnval(component_name,[C],Var2))

Figure 3. Canonical form produced by parser.

DAPLEX query:
   for each t in threeten
   for each c in structure_chain(t)
   print structure_name(t), component_name(c)

Grouped canonical form (gcf):
   getentity(threeten,T) # getfnval(structure_name,[T],Var1) * structure *
   getentity(chain,C) # getfnval(component_name,[C],Var2)

Where G# uses the binary infix operator # to group the generation of an entity with a list of the scalar functions on that entity.

E1*rel*E2 uses a ternary infix operator * to group a relationship with the entities which it relates. This operator is used as a handle by which the write rules can recognise relationships.

Figure 4. Example of group canonical form.

3.4.1 Keeping track of variables

A list of variables which currently have values is maintained during rewriting, because many operations can only be performed when values have been given to certain variables. This list is used by the low-level rewrite rule for expressions to ensure that all variables used in the expression have known values. For example, in the query

   for each p in protein such that resolution(p) < 2.5
   print protein_name(p);

in would be illegal to generate the reordered query:

   ... R < 2.5, getfnval(resolution, [P], R)...

because the variable R must be given a value before it is used in the comparison.

3.4.2 Use of inverses

There are two types of operation performed by the optimiser: the components of the gpf are reordered with respect to each other, and the individual components are themselves rewritten for more efficient execution. The reordering is controlled by the rewrite rule which generates permutations and by constraints imposed by unknown variables. For reordering to be successful it is necessary to make use of inverse functions.

The rewrite rule for inverses checks that the given function has an inverse, rewrites the result which is to become the argument, updates, the list of known variables, updates the score and rewrites the argument which is the result of the inverse. The rule can be read as:

   B1 <= rel(A1#A1R) can be written as
   A2#A2R <= relinv(B2) IF
   relinv is the inverse of rel AND
   B1 can be rewritten as B2 AND
   A1 is of the form generate(A2) AND
   A1R can be rewritten as A2R.

More formally, a term of the form E1 * f * E2, which uses a relationship f to find E2 given E1 can be rewritten as E1 * g * E2, where g is the inverse of f and used to find E1 given E2. Rewrite rules are applied recursively to E1 and E2, so that expressions are rewritten from the inside out. The Prolog form of this rule is given in the Appendix.

The reordering described up to this point is similar to the generation of strategies in Dayal, goal reordering in Warren and traversal generation in Gray. Other transformations performed by the optimiser and described below are concerned with the use of the index on the primary key, and the identification of duplicated calls in subqueries and boolean expressions.

3.4.3 Use of keys on entities

In P/FDM all entity classes have associated with them a (possibly compound) key constructed from scalar attributes of the entity class and the keys of related classes. An index on the key of a class provides direct access to
an entity given all the components of its key. In the gpf, all the restrict terms associated with an entity are grouped with the entity, and it is straightforward for a rewrite rule to examine these restrict terms for the components of the key. In the DAPLEX query below, the \textit{protein\_code} of the \textit{protein} is known. The key of an entity of class \textit{protein} is its \textit{protein\_code}. The key of a \textit{chain} is the concatenation of the key of the \textit{protein} to which it is linked by the function \textit{component\_protein} with the \textit{component\_id} of the \textit{chain}. Thus the routine \textit{getentity/3} introduced in Section 3.1 can be used to obtain the required \textit{chain} directly by key instead of by \textit{enumerate} and \textit{test} as implied by the for loop.

\textbf{DAPLEX:}

\begin{verbatim}
program p1 is
for each p in protein such that protein\_code(p) = "p2pab"
  for each c in chain such that component\_protein(c) = p and component\_id(c) = "B"
    print resolution(p), component\_name(c);
Optimised Prolog:

\texttt{p1:-
  ( getentity( chain, [p2pab, 'B'], C),
    getfnval( component\_name, [C], Var0 ),
    getfnval( component\_protein, [C], P ),
    getfnval( resolution, [P], Var1 ),
    write( [Var0,Var1] ), nl, fail
    ; true
  ).}

More formally, given a term of the form $E \# R$ in the gpf, which defines the restriction $R$ on the entity $E$, if the key of $E$ can be identified from $R$ and objects related to $E$, then the term can be rewritten as $E \# R$, where $E$ is a description of how to obtain $E$ by the index on its key, and $R$ is the remaining components of $R$.

3.3.4 Set operations

There are also rewrite rules which look for duplicated database calls which can be \textit{factored out}. For example, the query below which involves a \textit{union} operation can be rewritten with the union replaced by a disjunction.

\textbf{DAPLEX:}

\begin{verbatim}
for each S in ( S1 in structure such that length(S1) = 3 union
  S2 in structure such that length(S2) = 4 )
print structure\_name(S);
Direct mapping to prolog:

\texttt{\(\exists S1, S2.\)}
\texttt{( (findall(S1, (getentity(structure,S1),
    getfnval(length,[S1],3)), S1L ),
  findall(S2, (getentity(structure,S2),
    getfnval(length,[S2],4)), S2L ),
  union( S1L, S2L, UL),
  member(S, UL),
  getfnval( structure\_name, [S], Var1 ),
  write(Var1), nl, fail
  ; true
)).
\end{verbatim}

Optimised prolog:

\begin{verbatim}
( getentity(structure, S ),
  getfnval( length, [S], Var0 ),
  ( 3 ==: Var0 ; 4 ==: Var0 ),
  getfnval( structure\_name, [S], Var1 ),
  write( Var1 ), nl,
  fail
  ; true
 ).
\end{verbatim}

The rewriting in the query above is performed largely by two rules, one of which looks for common \textit{getentities} in a \textit{generate\_subquery}, while the other looks for common \textit{getfnvals} in a disjunction. When optimising relational algebra it is also possible to rewrite a union as a disjunction.

A selected\(_\text{on}[P1]\) projected\(_\text{to p1}\) + A selected\(_\text{on}[P2]\) projected\(_\text{to p1}\)

\[= A \text{selected\(_\text{on}[P1 \text{ or P2}\] projected\(_\text{to p1}\)\]}

A similar transformation takes place in the DAPLEX union, where the optimiser spots that both arguments of the union iterate over all instances of class \textit{structure}. The optimiser identifies the common \textit{generate} and renames the local variables in each of the arguments of the \textit{union}. The rewrite rule which performs this task is given in the Appendix.

In general, conjunctive and disjunctive expressions can be rewritten using the distributive rules given in Jarke and Koch,\cite{JarkeKoch} while union and intersection can be rewritten using:

\[A \# B \text{ union } A \# C \iff A \# (B \text{ or } C)\]
\[A \# B \text{ intersection } A \# C \iff A \# (B \text{ and } C)\]

3.4.5 Existentially quantified expressions

The quantified expressions of DAPLEX, such as \textit{some}, \textit{at least} and \textit{all} are supported by the optimiser as \textit{restrict\_subqueries}. For example, the Prolog produced for the DAPLEX in Fig. 3.2 is:

\begin{verbatim}
( getentity( chain, C ),
  atleast( 2, ( getfnval(structure\_chain\_inv,[C],Str),
    getfnval(level,[Str],4)) ),
  getfnval( cname, [C], Var0 ),
  write( Var0 ), nl,
  fail
  ; true
 ).
\end{verbatim}

The routine \textit{atleast/2} takes as arguments a count and a Prolog goal, and succeeds if the goal succeeds at least count times. The quantified expressions of DAPLEX jump out of the loop using the Prolog cut operator as soon as the condition is sure to succeed or sure to fail. For example, any \textit{restrict\_subquery} in which a variable is quantified by \textit{some} can exit successfully when a first value is found.\cite{JarkeKoch} The components of the \textit{restrict\_subquery} are also passed to the optimiser for rewriting so that strong restrictions are applied early (see Section 3.10).

3.5 Independent subqueries

When optimising queries expressed in logic against a relational database it is important to be able to identify
independent subqueries. Warren cites the following example query on a geographical database.

‘Which is the ocean that borders African countries and Asian countries?’
answer(X) <=
ocean(X) &
  { borders(X,C1) & african(C1) & country(C1) } &
  { borders(X,C2) & asian(C2) & country(C2) }.

The braces are inserted by the optimiser to group independent parts of the query. If the braces are omitted the last part of the query is needlessly repeated for each different African country which borders a given ocean. The same query expressed in DAPLEX looks quite different from the query in logic.

for the o in ocean such that
  some c1 in borders(o) has name(continent(c1)) = “Africa” and
  some c2 in borders(o) has name(continent(c2)) = “Asia”
print name(o);

In DAPLEX the subqueries are made explicit, and even in a system which performs no optimisation there is no combinatorial explosion because of the cut associated with the implementation of the quantifier some (see Section 3.4.5).

When performing updates in DAPLEX, meaningful subqueries can be identified. For example, the update make all first year science students take the course Maths1 is expressed in DAPLEX below:

program u1 is
  for each S in student such that
    year(S) = 1 and faculty(S) = “science”
  include takes(S) = { the C in course such that code(C) = “Maths1” }

If the statements are executed in the order in which they appear in the query, the subquery which retrieves the first-year maths course will be executed for every first-year science student. In our system the parser identifies independent subqueries in updates, has them optimised and ensures that they are written out before the body of the query.

u1:--
  ( getentity( course, [‘Maths1’], varl ),
    getentity( student, S ),
    getfnval( year, [S], 1 ),
    getfnval( faculty, [S], science ),
    addfnval( takes, [S], Var1 ),
    fail
  ; true
  )

3.7 Costing

The rewrite rules for optimisation generate successive versions of the given query by backtracking. These versions are evaluated, and the one with the lowest predicted cost is selected for conversion to Prolog. The predicted cost of a query is based upon the order of the DAPLEX loops. The cost of evaluating each successive level of nesting is the product of the number of times that level is expected to be executed and the cost of executing it.

global_cost₀ = 0
global_cost_n = global_cost_n-1 + num_times_n * local_cost_n

where n ranges from 0 to the total depth of nesting.

The number of times the statements at a level are executed is given by:

num_times₀ = 1
num_times_n = num_times_n-1 * (level_num_times_ / level_restrictivity_n)

The value level_num_times_n is the cardinality of the principal generate at each level, and is taken from an average value stored in the database. The term level_restrictivity_n is an indication of the strength of constraints present at level n. The cost of evaluating each level in a program is worked out from the estimated cost and restrictivity of each generate and restrict term at that level.

local_cost₀ = 0
local_cost_m = local_cost_m-1 + (item_cost_m/restrictivity_m)
restrictivity₀ = 1
restrictivity_m = restrictivity_m-1 * item_restrictivity_m

where m ranges from 0 to the number of components at the current level.

The item_restrictivity (an integer >= 1) is an estimate of the restrictivity of each item at the current level.

3.8 Example of Costing

In this section details are given of the output from the example query given below.

for each p in protein
  for each c in component_protein such that
    component_protein(c) = p and component_name(c) = “B”
print protein_code(p), component_id(c);

The basic operations in P/FDM notation are given below.

(1) getentity( component_protein, C )
(2) getfnval( component_id, [C], Cid )
(3) getfnval( component_name, [C], ‘B’ )
(4) getfnval( component_protein, [C], P )
(5) getfnval( protein_code, [P], PCode )
(6) getfnval( component_protein_inv, [P], C )
(7) getentity( protein, P )

The optimiser reorders these operations, estimating the cost of each. Six ways of evaluating the query are generated, the orders of which are given below along with estimated costs and actual cpu times. Two of the generated solutions make use of the inverse of the
function component_protein to generate entities of class protein_component when the protein is used as the starting point for the search.

<table>
<thead>
<tr>
<th>Order</th>
<th>Predicted cost</th>
<th>Cpu time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,2,3,4,5</td>
<td>88</td>
<td>64</td>
</tr>
<tr>
<td>1,3,2,4,5</td>
<td>54</td>
<td>53</td>
</tr>
<tr>
<td>7,5,6,2,3</td>
<td>128</td>
<td>67</td>
</tr>
<tr>
<td>7,5,6,3,2</td>
<td>94</td>
<td>56</td>
</tr>
<tr>
<td>1,2,3,7,5,6</td>
<td>3368</td>
<td>578</td>
</tr>
<tr>
<td>1,3,2,7,5,6</td>
<td>3334</td>
<td>574</td>
</tr>
</tbody>
</table>

The results show the relationship between the predicted cost and the actual time taken. The order with the lowest predicted cost is the quickest to execute, and the slowest ‘generate and test’ evaluation methods are predicted to be expensive. The optimiser has clearly managed to avoid the O(n^3) approaches to evaluating this query and has given a plausible ordering to the O(n) strategies. The time taken to parse and optimise the example query using Quintus Prolog on a Sun 3/75 workstation is 2.8 s.

3.9 Justification of methodology

The nature of the operations performed by the optimiser is straightforward, and can be seen from a number of perspectives. If a query is considered to be a series of constraints upon a network of objects, these can be viewed as a conjunction of terms such as:

(3 S,C) student(s) & takes(S,C) & course(C)

This query can then be rewritten using the fact that ‘&’ is commutative.8 Similar arguments can be applied to relational algebra transformations as described in Gray.9

A common difficulty with systems using rewrite rules is that they can find themselves in a loop. For example, A * B is rewritten as B * A which is rewritten as A * B and so on. The problem arises when, as is commonly the case when transforming relational algebra queries, a rewritten expression is itself rewritten. In the optimiser described here, cycles do not occur because alternatives are generated by backtracking to alternative rules. This compares favourably with a recursive approach in which the entire rewritten expression is likely to be reoptimised, thereby running the risk of having earlier transformations undone.

3.10 Reducing the search space

When generating alternative access paths, the optimiser explores the search space of legal ways to evaluate a query. For substantial queries this search space can be rather large, and the time taken for optimisation can become significant. A number of steps are taken to prune the search space.

The optimal way of evaluating the body of a subquery is independent of the place in the query in which the subquery is evaluated. Thus it is possible to rewrite all subqueries once before rewriting the body of the query. All subqueries are analysed and rewritten during the first pass of the optimiser (Fig. 2). During the rest of the optimisation they are considered as a black box associated with a list of required variables, a cost and a restrictivity.

During preprocessing, components are inserted in the gpf such that generates with strong constraints are selected early, an approach similar to the application of the cheapest first heuristic of Smith and Genesereth.26

This means that the best way of evaluating a query is likely to be one of the first produced by the optimiser. The routine which builds the global cost (see Section 3.6) fails whenever the score to which it is adding becomes greater than the best score for the whole query produced up to that point. With this tree trimming, only two of the six possible ways of evaluating the query in Section 3.8 are generated by the optimiser.

3.11 Optimiser code

The implementation of the optimiser consists of approximately 1400 lines of Prolog, while the DAPLEX parser is approximately 1500 lines. An earlier version of the optimiser, also in Prolog but not based upon rewrite rules, was approximately 1500 lines as well as being slower and less powerful than the present version. There are currently 20 rewrite rules in the optimiser. This and previous work on the use of Prolog to optimise relational algebra queries8 indicates that Prolog permits rapid prototyping of concise and maintainable systems which can be compiled to give reasonable performance.

4. FUTURE WORK

4.1 Optimising Methods

In an object-oriented database, the question arises of how to give a score to the code of methods stored in the database. Methods may be associated with new types of data or non-standard storage structures.

The obvious way of giving a score to a method is to store the score assigned to it by the optimiser with the method in the database when the method is defined. The advantages of this approach are that it gives a reasonable estimate of the cost of evaluating a method and takes account of one method calling others. One disadvantage is that methods are not invertible, and thus the scope the optimiser has for rewriting the complete query is reduced.

A more ambitious approach is to use partial evaluation30 to recursively copy the canonical form of the method into the canonical form of the query. The query, with the primitive calls of all methods in-line, could then be passed to the optimiser for rewriting. This approach has the advantage that methods will be inverted if this is possible and that query optimisation will be truly global. Related proposals are made by Graefe and Maier7 and Zdonik,32 who suggest that the query optimiser, as a trusted part of the system, should be able to ask a method to reveal information on its execution strategy.

The price to be paid for the more thorough optimisation is that the canonical form to be optimised could become rather large, and thus the time taken for optimisation could become significant when compared with the time taken for query execution. The ideal solution would be to have both modes of optimisation available so that the user can choose between them. The general heuristic for choosing between the two approaches is: for large databases with simple methods choose thorough optimisation, for small databases with complex methods choose quick optimisation.

4.2 Improving the optimiser

New rules can be added to the optimiser very easily to increase the range of transformations it can make. The
definition of DAPLEX does not permit recursive queries to be defined over the extension of the database, so the optimiser does not implement a strategy for processing recursive queries. P/FDM is being used with protein structure data, and it is likely that this application will require extensions to DAPLEX to permit the expression of recursive queries.

While the optimiser has been written as a back end to a parser for DAPLEX, the parser and optimiser are distinct units. A QBE-like front end is being developed which outputs the canonical form described in Section 3.2.

5. CONCLUSIONS

We have built an object-oriented database for handling protein structures primarily because we needed to represent both amino acid sequences and a rich variety of data types. A query language based upon DAPLEX has been implemented which allows scientists to use a navigational style of query, iterating over entity sets and ranges of integers, and which supports calls to methods and procedures stored in the database. Nevertheless, this style of query is only practical with an optimiser (actually, a query improver) which rewrites the query to use sensible strategies.

This paper shows how such queries can be optimised by treating them as conjunctive predicate calculus expressions including subqueries which possibly contain disjunctions. The optimiser is conveniently structured as Prolog rules for generating Prolog intermediate code, which in turn calls out to persistent storage routines and stored methods. This architecture interfaces neatly with the Prolog parser and shows that Prolog is not confined to use with relational data. The optimiser handles a navigational style of query, including a full range of DAPLEX constructs, and we believe this style suits object-oriented databases better than the set-based style of SQL. It also shows promise for extensions which use cost information on methods.

Acknowledgement

This work is partly supported by grant GR/D/32765 from the UK Science and Engineering Research Council.

REFERENCES

APPENDIX

As described in Section 3, rewrite rules are a mechanism for replacing one representation of a structure with some alternative equivalent representation. The Prolog rewrite rules used by the optimiser estimate the cost of legal permutations of the elements in the grouped P/FDM form (gpf) of the query. The rewrite rules have six arguments. The first three pass data into the rule (the input gpf, a list of known variables and the cost so far) and the last three pass data out of the rule (the rewritten gpf, the variables known after executing the gpf and the cost after executing the gpf).

The rewrite rule for inverses checks that the given function has an inverse, rewrites the result which is to become the argument, updates the list of known variables, updates the cost and rewrites the argument which is the result of the inverse. This rule in Prolog is:

\[
\text{rewrite( A1Gen#A1Res*Rel*B1, InVars, InCost, }
\text{ B2*RelInv*Result*A2Res, OutVars, OutCost):-}
\]

\[
\% \text{ Invert the given function}
\]

\[
\% \text{ has_fdesc( Rel, [ArgType], ResType ),}
\text{ has_inverse( Rel, RelInv ),}
\text{ has_fdesc( RelInv, [ResType], ArgType ),}
\text{ rewrite( B1, InVars, InCost, B2, MidVars1, MidCost1 )},
\]

\[
\text{ generate_result( A1Gen, Result ),}
\text{ add_variable( Result, MidVars1, MidVars2 ),}
\]

\[
\text{ ... rewrite( A1Res, MidVars2, MidCost2, A2Res, OutVars, OutCost ) .}
\]

When rewriting the DAPLEX `union` operator, the optimiser identifies the common `getentity` and renames the local variables in each of the arguments of the `union`. The rewrite rule which performs this task is given below.

\[
\text{rewrite( generate_subquery(C,Var,union(V1,V2),}
\text{ [getentity(C,V1)#R1],[getentity(C,V2)#R2])#RIn,}
\text{ InVars, InCost, OutCode, OutVars, OutCost):-}
\]

\[
\% \text{ Rewrite restrictions to use common variable Var}
\text{ rename(R1, V1, Var, RenameR1),}
\text{ rename(R2, V2, Var, RenameR2),}
\text{ join_expr( [ RenamedR1 ; RenamedR2], RIn, Res ),}
\text{ rewrite( getentity(C,Var)#Res, InCost, OutCode,}
\text{ OutVars,OutCost ) .}
\]

The rename routine is also implemented using rewrite rules.