Decomposition—A Strategy for Query Processing

EUGENE WONG AND KAREL YOUSSEFI
University of California, Berkeley

Strategy for processing multivariable queries in the database management system INGRES is considered. The general procedure is to decompose the query into a sequence of one-variable queries by alternating between (a) reduction: breaking off components of the query which are joined to it by a single variable, and (b) tuple substitution: substituting for one of the variables a tuple at a time. Algorithms for reduction and for choosing the variable to be substituted are given. In most cases the latter decision depends on estimation of costs; heuristic procedures for making such estimates are outlined.

Key Words and Phrases: relational database, query processing, decomposition, tuple substitution, detachment, joining (overlapping) variable, irreducible query, connected query, variable selection
CR Categories: 3.50, 3.70, 4.3

1. INTRODUCTION

The structural simplicity of a relational data model encourages the use of a nonprocedural data sublanguage which specifies what is to be found rather than how it is to be found. Thus it is not surprising that nearly every one of the relational languages which have been proposed is nonprocedural. As is generally true with high level languages, a price which may have to be paid is a loss of efficiency. For a relational database of any size and for queries spanning several relations the price can be fearsome. Results of various degrees of generality on improving search strategies for a relational database system have been reported by Palermo [6], Astrahan and Chamberlin [2], Rothnie [10, 11], Pecherer [7], Smith and Chang [12], and Todd [14]. Nonetheless the lack of a general approach to optimizing query processing remains a major impediment to achieving a satisfactory degree of efficiency for nonprocedural relational languages.

The purpose of this paper is to describe in some detail the query processing algorithm developed for QUEL [4], which is the data language for the INGRES sys-

A version of this paper was presented at the ACM SIGMOD International Conference on Management of Data, Washington, D.C., June 2-4, 1976.
This research was sponsored by the US Army Research Office—Durham under Grant DAHCO4-74-G0087 and the Joint Services Electronics Program under Contract F44620-71-C-0087.
Authors' address: Department of Electrical Engineering and Computer Sciences, Electronics Research Laboratory, University of California, Berkeley, Berkeley, CA 94720.

tem. Insofar as the problems encountered in QUEL are common to all nonprocedural relational languages, their solution should find general application.

In Section 2 a brief description of QUEL, the query language to be processed, is presented. In Section 3 we sketch a skeletal outline of the decomposition algorithm emphasizing the functions of the component algorithms and the flow of information and control among them. The details of the component algorithms are presented in subsequent sections.

2. QUEL

A complete definition of QUEL is given in [4]. Here we shall confine ourselves to a brief description sufficient to make the processing strategy comprehensible. There are four commands: RETRIEVE, REPLACE, DELETE, APPEND. An update command is turned into a RETRIEVE command which is then followed by a low level tuple-by-tuple operation. We shall restrict our attention to RETRIEVE. A statement to retrieve in QUEL has the following form:

```
RANGE OF {Variable} IS {Relation}
RETRIEVE INTO Result-Name (Target-List) WHERE Qualification
```

Example 2.1. Consider a database with relations

```
Supplier (S#, Sname, City)
Parts (P#, Pname, Size)
Supply (S#, P#, Quantity)
```

and a query to find the names of all parts supplied by suppliers in New York. This can be stated in QUEL as follows:

```
RANGE OF (S,P,Y) IS (Supplier, Parts, Supply)
RETRIEVE INTO NYparts (P.Pname) WHERE (P.P#=Y.P#)
                                AND   (Y.S.#=S.S#)
                                AND   (S.City='New York')
```

From the point of view of query processing there are two principal sources of complexity. First, QUEL permits aggregation operators such as MAX and AVG with nesting of such operators. Second, queries involving several variables require deft handling in order to avoid the obvious possibility of combinatorial growth. For example, if the query in Example 2.1 is processed by first forming a cartesian product, then the number of tuples to be scanned is equal to the product of the cardinalities of the three relations. In our system all aggregations are performed on single relations. If an aggregation is to be done on a subset of the product of several relations, the subset must first be assembled by processing a multivariable query. Aggregations once evaluated are kept for possible reuse until updates render them obsolete. In the remainder of the paper we shall deal only with aggregation-free queries, and the thrust of the query processing strategy is to cope effectively with aggregation-free but multivariable queries.

Let $X = (X_1, X_2, \ldots, X_n)$ denote the variables declared in the range statement, and let $R_1, R_2, \ldots, R_n$ be their respective ranges. Then the qualification can be considered to be a boolean function $B(X)$ on the cartesian product $R = R_1 \times R_2 \times \ldots \times R_n$. The target list can be considered to be a set of functions $(T_1(X), T_2(X), \ldots, T_m(X)) = T(X)$ on the product space, and the result relation of the query is constructed by evaluating T(X) on the subset of R defined by B(X) = 1, and eliminating duplicate tuples. We note that for a query free of aggregation operators each tuple X in the product space R contains enough information to completely determine the values of B(X) and T(X).

The interpretation of QUEL statements suggests the following procedure for their processing:

(a) Product: A cartesian product of the range relation is formed.

(b) Restriction: Tuples X in the product which satisfy B(X) = 1 are determined.

(c) Computation and projection: T(X) is computed on the subset determined in (b) and duplicate tuples are eliminated.

Unfortunately this procedure is as inefficient as it is obvious. The cardinality of the product R, i.e. the number of tuples in R, is equal to the product of the cardinalities of R_i, $i = 1, 2, \ldots, n$. It does not take very large relations or very many of them to make this number enormous. Aside from the difficulty of having to form and store a very large relation, to determine the subset which satisfies B(X) = 1 requires examining a number of tuples equal to the cardinality of R.

3. DECOMPOSITION

The query processing strategy that we have adopted has two overall objectives:

(a) No cartesian product: The result relation is to be constructed by assembling comparatively small pieces rather than by paring down the cartesian product.

(b) No geometric growth: The number of tuples to be scanned is to be kept as small as possible; for most queries this number is much less than the cardinality of R.

Our general procedure is to reduce an arbitrary multivariable query to a sequence of single-variable ones. We call this process *decomposition*. Observe that the first objective is automatically achieved by such an approach. To attain the second requires a detailed examination of the tactical moves which are available.

The decision to reduce multivariable queries to single-variable ones separates the overall optimization into two levels. It has obvious advantages in structuring the optimization procedure, which otherwise may well become unbearably complex. The only situation in which our approach may be undesirable is when interrelational information such as "links" [15] is available, in which case the desirable atomic units may be two-variable queries.

It is useful to distinguish two types of operations which are repeatedly invoked in decomposition:

(a) *Tuple substitution*: An n-variable query Q is replaced by a family of (n − 1)-variable queries resulting from substituting for one of its variables tuple by tuple, i.e.

$$Q(X_1, X_2, \ldots, X_n) \to \{Q'_\alpha(X_2, X_3, \ldots, X_n), \alpha \in R_1\}.$$

(b) *Detachment* of a subquery with a single overlapping variable: A query Q is replaced by Q′ followed by Q″ such that Q′ and Q″ have only a single variable in common.

Operations of these two types suffice to decompose any query completely. Indeed, a series of successive tuple substitutions is sufficient, albeit tantamount to forming the cartesian product. Tuple substitution for a single variable means that the cost of processing the remaining portion of the query is multiplied by a factor which in most cases is equal to the cardinality of the range of the substituted variable. It is important, therefore, that the ranges of the variables be reduced as much as possible before substitution takes place. The most straightforward way of doing this is through restriction and projection, which are special cases of detachment. Something equivalent to such a step has been proposed in every paper on optimizing query processing.

Example 3.1. Consider a database with three relations:

```
Supplier (S#, Sname, City)
Parts (P#, Pname, Size)
Supply (S#, P#, Quantity)
```

and a query Q:

```
RANGE OF (S,P,Y) IS (Supplier, Parts, Supply)
RETRIEVE (S.Sname) WHERE (S.City='New York')
                    AND   (P.Pname='Bolt')
                    AND   (P.Size=20)
                    AND   (Y.S#=S.S#)
                    AND   (Y.P#=P.P#)
                    AND   (Y.Quantity≥200)
```

The first detachment might be a restriction on Parts resulting in Q being replaced by

```
Q1  RANGE OF (P) IS (Parts)
    RETRIEVE INTO Parts1 (P.P#) WHERE (P.Pname='Bolt')
                                AND   (P.Size=20)
Q′  RANGE OF (S,P,Y) IS (Supplier, Parts1, Supply)
    RETRIEVE (S.Sname) WHERE (S.City='New York')
                       AND   (Y.S#=S.S#)
                       AND   (Y.P#=P.P#)
                       AND   (Y.Quantity≥200)
```

Let us represent this detachment by a binary tree:

$$\begin{tikzcd}
 & Q \arrow[dl] \arrow[dr] & \\
Q' & & Q1
\end{tikzcd}$$

Then the successive detachment operations on Q can be represented by

```
Q
├──→Q1  :  (P.P#) WHERE (P.Size=20) AND (P.Pname='Bolt')
├──→Q2  :  (Y.P#, Y.S#) WHERE (Y.Quantity≥200)
├──→Q3  :  (S.S#, S.Sname) WHERE (S.City='New York')
├──→Q4  :  (Y.S#) WHERE (Y.P#=P.P#)
↓
Q5  :  (S.Sname) WHERE (Y.S#=S.S#)
```

In this example detachment operations have reduced Q to three one-variable queries Q1, Q2, Q3 which can be processed in parallel or in arbitrary order, followed by a two-variable query Q4 and then another two-variable query Q5. Q4 and Q5 cannot be further reduced by detachment operations, and tuple substitution must be used to complete the decomposition. We note, however, that the ranges of the variables in Q4 and Q5 are likely to be very much smaller than the original relations, and tuple substitution at these stages is relatively harmless. As an example of tuple substitution, consider

```
Q5 : RETRIEVE (S.Sname) WHERE (Y.S#=S.S#)
```

Suppose that at this point the range of Y is the relation

S#
101
107
203

Then, successive substitution of Y yields

```
Q5(101) : RETRIEVE (S.Sname) WHERE (S.S#=101)
Q5(107) : RETRIEVE (S.Sname) WHERE (S.S#=107)
Q5(203) : RETRIEVE (S.Sname) WHERE (S.S#=203)
```

We note that unlike SEQUEL [2], QUEL has no sequential structure and there is no a priori preferential order of substitution for the variables.

The general situation covered by the detachment operation is the following: Consider a query of the form:

$$\begin{array}{ll} & \text{RANGE OF } (X_1, X_2, \ldots, X_n) \text{ IS } (R_1, R_2, \ldots, R_n) \\ Q & \text{RETRIEVE } T(X_1, X_2, \ldots, X_m) \\ & \text{WHERE } B''(X_1, X_2, \ldots, X_m) \\ & \text{AND} \quad B'(X_m, X_{m+1}, \ldots, X_n) \end{array}$$

It is natural to break off B' to form

$$\begin{array}{ll} & \text{RANGE OF } (X_m, X_{m+1}, \ldots, X_n) \text{ IS } (R_m, R_{m+1}, \ldots, R_n) \\ Q' & \text{RETRIEVE INTO } R_m'(T'(X_m)) \\ & \text{WHERE } B'(X_m, X_{m+1}, \ldots, X_n) \end{array}$$

where $T'(X_m)$ contains the information on X_m needed by the remainder of the query, which can now be expressed as

$$\begin{array}{ll} & \text{RANGE OF } (X_1, X_2, \ldots, X_m) \text{ IS } (R_1, R_2, \ldots, R_m') \\ Q'' & \text{RETRIEVE} \quad T(X_1, X_2, \ldots, X_m) \\ & \text{WHERE} \quad B''(X_1, X_2, \ldots, X_m) \end{array}$$

Observations: (1) Q'' is necessarily simpler than the original query Q since $m \leq n$ and R_m' is smaller than R_m. Even for the worst possible case where $R_m' = R_m$ and $m = n$, Q'' is no worse than Q. (2) The detachment of Q' does not lead to an increase in the maximum number of variables for which substitution has to be made. To see this, note that the maximum number of variables to be substituted for in an n-variable query is $n - 1$. Hence this number is $(n - m + 1) - 1$ for Q' and $m - 1$ for Q'' so that the total is again $n - 1$. (3) Q' and Q'' are strictly ordered. Q' needs no information from Q'', so it can be processed completely before processing on Q'' begins. At any given time we only need to deal with a total of n or less variables.

Two special cases of one-overlapping-variable subqueries are worthy of special note. First it may happen that the detached subquery Q' has no variable in common with the remainder Q''. That is, B' is a function of only $(X_{m+1}, \ldots, X_n)$ and not of X_m. In such a case we say Q' is a *disjoint* subquery. The interpretation of this situation is that if B' is satisfied by a nonempty set, then Q is equivalent to Q'', otherwise Q is itself void, i.e. its result is empty. The second special case arises when $m = n$ and B' is a one-variable query. This is a frequent and important occurrence, as the previous example illustrates. We say a query is *connected* if it has no disjoint subquery, *one-free* if it has no one-variable subquery, and *irreducible* if it has no one-overlapping-variable subquery. An irreducible query is obviously both connected and one-free.

Broadly speaking, we will always break up a query into irreducible components before tuple substitution. In effect we will always prefer not to tuple substitute if we can avoid or postpone it. Although examples can be constructed to show that such a choice is not always optimal, in general this is not a bad heuristic. Detaching subqueries involves an additive growth in complexity, while tuple substitution incurs a multiplicative growth. Our decomposition algorithm is recursively applied to all the subqueries which are generated.

4. A DESCRIPTION OF THE ALGORITHM

The *decomposition algorithm* consists of four subalgorithms, *reduction*, *subquery sequencing*, *tuple substitution*, and *variable selection*, and makes use of the one-variable processor of the system. The interaction among these component processes is indicated in Figure 1. The fact that the decomposition algorithm is recursive is made clear by the fact that decomposition calls itself. The basic functions of the subalgorithms are as follows:

(a) *Reduction* breaks up the query into irreducible components and puts them in a certain sequential order.

(b) *Subquery sequencing* uses the result of reduction and generates in succession subqueries each of which contains a single irreducible component together with one-variable clauses. As each subquery is generated it is passed to tuple substitution, and the generation of the next subquery awaits return of the result.

(c) *Tuple substitution* manages the process of substituting tuple values. It calls variable selection to select a single variable for substitution. After substituting each tuple for that variable, it passes the resulting reduced query to Reduction and awaits the return before substituting the next value.

(d) *Variable selection* is where most of the optimization takes place. It estimates the relative cost of substituting for each variable and chooses the variable with the minimum estimated cost. In so doing, it may have to preprocess some one-variable subqueries.

The details of the subalgorithms will be described in the next few sections.

4.1 Reduction Algorithm

The input consists of a multivariable query Q, and the output consists of the irreducible components of Q arranged in an appropriate sequential order. This sequence is passed to subquery sequencing, and the result relation for Q is returned. The basic steps of the algorithm are illustrated in Figure 2.

Let $X = (X_1, X_2, \ldots, X_n)$ denote the variables of Q and let $T(X)$ and $B(X)$

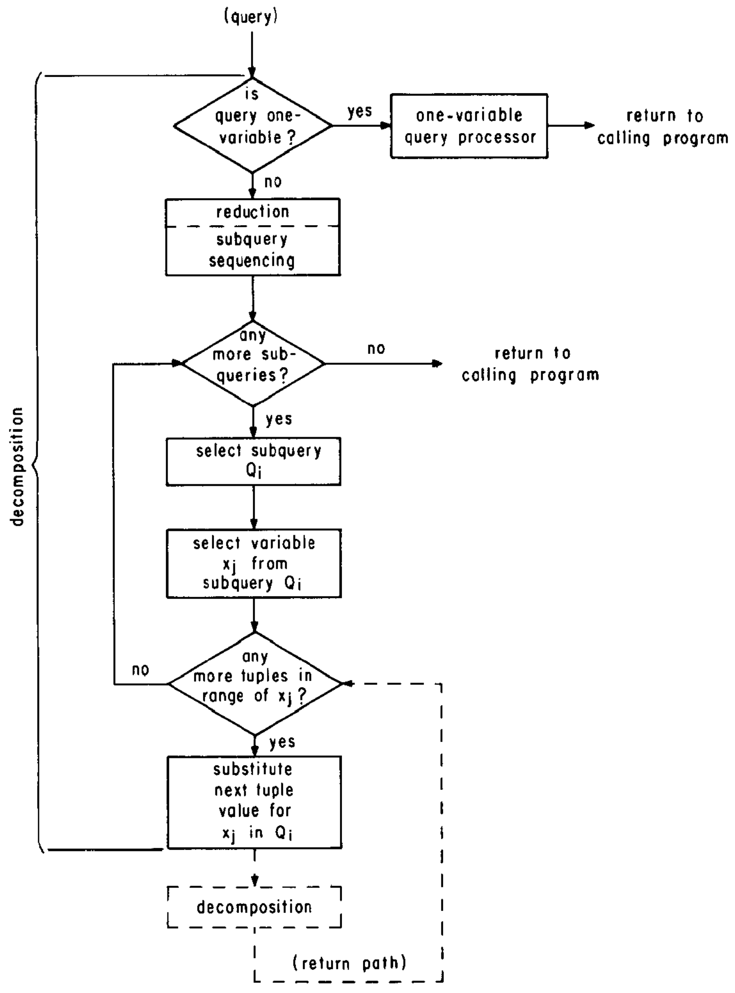

(query)
is query one-variable ?
yes
one-variable query processor
return to calling program
no
reduction
subquery sequencing
any more sub-queries?
no
return to calling program
yes
select subquery Qi
select variable xj from subquery Qi
any more tuples in range of xj ?
no
yes
substitute next tuple value for xj in Qi
decomposition
(return path)
decomposition

Fig. 1. Flow of control in decomposition

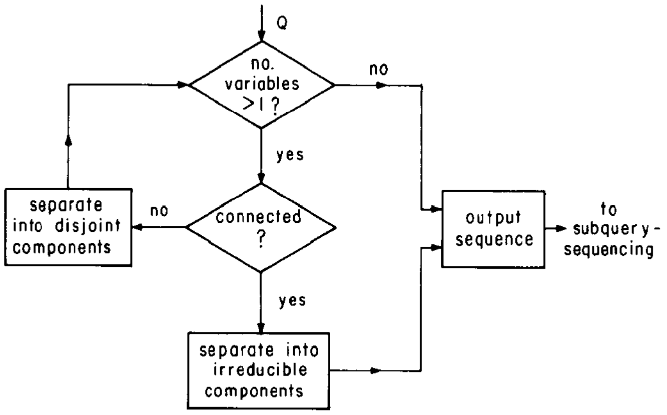


Fig. 2. Reduction algorithm

denote its target list and qualification, respectively. We assume that B(X) is expressed in conjunctive normal form

$$B(X) = \bigwedge_i C_i(X),$$

where each clause $C_i(X)$ contains only disjunctions. Now consider a binary (0 or 1) matrix with p + 1 rows corresponding to T(X) and the p clauses, and with n columns corresponding to the variables $X_1, \ldots, X_n$. An entry of 1 denotes the presence of a variable in a clause (or target list), and 0 denotes its absence. We call this the *incidence matrix*. For Example 3.1 this matrix is given by

		S	P	Y
T:	S.Sname	1	0	0
C1:	S.City='New York'	1	0	0
C2:	P.Pname='Bolt'	0	1	0
C3:	P.Size=20	0	1	0
C4:	Y.S#=S.S#	1	0	1
C5:	Y.P#=P.P#	0	1	1
C6:	Y.Quantity≥200	0	0	1

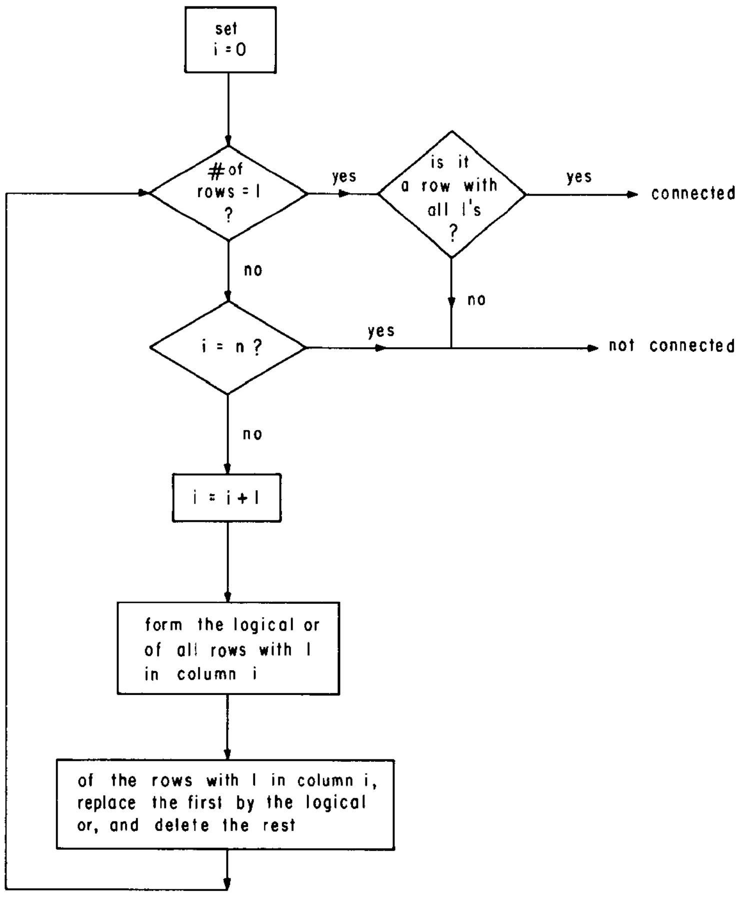

set i = 0
# of rows = 1 ?
yes
is it a row with all 1's ?
yes
connected
no
no
i = n ?
yes
not connected
no
i = i + 1
form the logical or of all rows with 1 in column i
of the rows with 1 in column i, replace the first by the logical or, and delete the rest

Fig. 3. Connectivity

We note that in Figure 1 there are two steps for which detailed algorithms remain to be provided. First we need a test for connectedness and to separate Q into disjoint components if it is not connected. Second we need an algorithm to separate a connected query into irreducible components and to put them in a suitable sequential order.

(a) *Connectivity algorithm.* If the connectivity algorithm (Figure 3) results in a matrix with a single row which is not all 1's, then the variables corresponding to the zero entries are superfluous and can be eliminated. If the final matrix has more than one row, then the sets of variables corresponding to different rows must be disjoint. If we keep track of the original rows which are combined to make up each of the rows of the final matrix, then the connected components of the query can be separated.

Consider Example 3.1, modified by the deletion of C4. The incidence matrix now has the form:

	S	P	Y
T	1	0	0
C1	1	0	0
C2	0	1	0
C3	0	1	0
C5	0	1	1
C6	0	0	1

Applying the connectivity algorithm, we get successively:

	S	P	Y
T, C1	1	0	0
C2	0	1	0
C3	0	1	0
C5	0	1	1
C6	0	0	1

	S	P	Y
T, C1	1	0	0
C2, C3, C5	0	1	1
C6	0	0	1

	S	P	Y
T, C1	1	0	0
C2, C3, C5, C6	0	1	1

Hence the query is not connected and the connected components are (T, C1) and (C2, C3, C5, C6).

(b) *Reduction into irreducible components.* Let Q be a connected multivariable query. We observe that it is reducible if the elimination of any one variable results in Q being disconnected. Let a variable with this property be called a *joining variable.* Thus, Q is irreducible if and only if none of its variables is a joining variable. Joining variables have some important properties which greatly facilitate the reduction algorithm, and these are summarized as follows:

PROPOSITION 4.1. *Suppose that X is a joining variable of Q such that its removal disconnects Q into k connected components. Then any joining variable of one of the components is a joining variable of Q, and every joining variable of Q is a joining variable of* *one of the components. Further, successive elimination of two joining variables in either order results in reducing Q to the same disjoint components.*

PROOF. Each joining variable joins a number of components which can overlap only on the joining variable. Let X be a joining variable of Q which joins components $Q_1, Q_2, \ldots, Q_k$. Let Y be a joining variable of one of these components, say Q_1. Then Y joins components $Q_{11}, Q_{12}, \ldots, Q_{1j}$ of Q_1, only one of which can contain X, say Q_{11}. Therefore, $(Q_{12}, \ldots, Q_{1j})$ overlaps the remainder of Q only on Y, and Y is a joining variable of Q. Conversely, let Y be a joining variable of Q, and join components $Q_1', Q_2', \ldots, Q_j'$. Only one of the set $\{Q_1', Q_2', \ldots, Q_j'\}$ can contain X, say Q_1', and only one of the set $\{Q_1, Q_2, \ldots, Q_k\}$ can contain Y, say Q_1. Then $\{Q_2', \ldots, Q_j'\}$ and $\{Q_2, \ldots, Q_k\}$ must be disjoint since each Q_i, $i \geq 2$, can overlap its remainder in Q only on X and none of $\{Q_2', \ldots, Q_j'\}$ contains X. Hence, $Q_2', \ldots, Q_j'$ are subsets of Q_1 joined to it only by Y, so that Y is a joining variable of Q_1. It is clear that Q has components $\{Q_2, Q_3, \ldots, Q_k\}$ each joined by only X, $\{Q_2', Q_3', \ldots, Q_j'\}$ each joined by only Y, and Q_{xy} joined by both X and Y. Elimination of X and Y in either order results in disjoint components $\{\bar{Q}_2, \bar{Q}_3, \ldots, \bar{Q}_k, \bar{Q}_2', \ldots, \bar{Q}_j', \bar{\bar{Q}}_{xy}\}$, where $\bar{Q}_i$ denotes Q_i with X removed, $\bar{Q}_i'$ denotes Q_i' with Y removed, and $\bar{\bar{Q}}_{xy}$ denotes Q_{xy} with both X and Y removed.

The substance of Proposition 4.1 is illustrated by Figure 4.

The results of Proposition 4.1 mean that we can find the irreducible components of Q by successively checking each variable for the possibility of its being a joining variable. Each variable only needs to be examined once, and the order of testing is immaterial. Further, since a variable is joining if and only if its elimination disconnects Q, we can use the connectivity algorithm for the test.

Take the incidence matrix of Q and eliminate from it all rows with only a single 1. Beginning with the first, eliminate each column in turn and test for connectedness. Suppose that when column m is eliminated Q breaks up into k connected components with $n_1, n_2, \ldots, n_k$ variables, respectively. Then these correspond to components of Q with $n_1 + 1, n_2 + 1, \ldots, n_k + 1$ variables, respectively, any pair of which overlap only on X_m. We can now proceed to test columns $m + 1, \ldots, n$. We note that each of the variables $X_{m+1}, \ldots, X_n$ occur in only one of the components so that after the m-th column (i.e. the first joining variable) the tests are performed on matrices of reduced size.

Each irreducible component of Q corresponds to one or more rows of the incidence matrix and can be represented by the "logical or" of the corresponding rows. Hence Q can be represented in terms of its irreducible components by a matrix with variables as columns and components as rows. We shall call this the *reduced-incidence matrix*. It is convenient to arrange the rows as follows:

(1) One-variable rows except the target list.

(2) Components which are one-overlapping after deletion of one-variable clauses

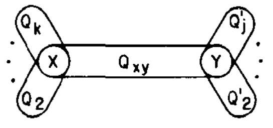


Fig. 4. Joining variables

and which do not contain the target list. These should be grouped according to the joining variable.

(3) Other components which do not contain the target list.

(4) The component which contains the target list.

For Example 3.1 the resulting reduced incidence matrix is given by:

	S	P	Y
C1	1	0	0
C2	0	1	0
C3	0	1	0
C6	0	0	1
C5	0	1	1
T, C4	1	0	1

4.2 Subquery Sequencing

The task of this program is simple. It receives the output of Reduction and forms a subquery by taking the component corresponding to the first multivariable row of the reduced-incidence matrix and combining it with all one-variable clauses in the same variables. It deletes the rows which have been used and passes the subquery to Tuple Substitution. Upon return of the result of the subquery, it repeats the process on the remaining matrix until the matrix is exhausted and the result of Q is returned. It then returns the result of Q to the calling program.

For Example 3.1 the subqueries which get generated are as follows:

```
Q1 : C2, C3, C6, C5
Q2 : C1, C4, T
```

More explicitly, we have

```
Q1 : RANGE OF (P,Y) IS (Parts, Supply)
     RETRIEVE INTO Supply1 (Y.S#) WHERE (P.Pname='Bolt')
                                  AND   (P.Size=20)
                                  AND   (Y.Quantity≥200)
                                  AND   (Y.P#=P.P#)
Q2 : RANGE OF (S,Y) IS (Supplier, Supply1)
     RETRIEVE (S.Sname) WHERE (S.City='New York')
                        AND   (Y.S#=S.S#)
```

4.3 Tuple Substitution

The input to tuple substitution is a query Q consisting of a single irreducible component in variables $X_1, X_2, \ldots, X_n$, zero or more one-variable clauses in each of the variables, and the range relations $R_1, R_2, \ldots, R_n$ of the variables. It returns the result relation to the calling program.

The first thing that tuple substitution does is to call variable selection which takes Q and the range relations and chooses a variable to be substituted for. In order to make this choice it may have to process some or all of the one-variable clauses to restrict the ranges. Thus, in general, it returns $\{Q', R_1', R_2', \ldots, R_n'\}$ and the variable to be substituted for (say X_n). For each tuple α in R_n', Q' becomes an $(n - 1)$-variable query $Q'(\alpha)$ in the variables $X_1, X_2, \ldots, X_{n-1}$. For each α, $Q'(\alpha)$ is passed to reduction, which returns the result. The results to $Q'(\alpha)$ for all α in R_n' are accumulated and returned to the calling program.

5. VARIABLE SELECTION

This is the heart of optimization. The input is a multivariable query which is irreducible except for one-variable clauses. As its name suggests, the task of this portion of the decomposition algorithm is to select a variable for substitution, although to do so it may also have to process some of the one-variable clauses.

Consider a query Q with variables $X_1, X_2, \ldots, X_n$ and ranges $R_1, R_2, \ldots, R_n$. Suppose that X_i is substituted tuple by tuple. For each tuple Q becomes an $(n - 1)$-variable query $Q_i(\alpha)$. It is likely that $Q_i(\alpha)$ takes the same amount of time to process for every α, and in most instances every α in R_i has to be used. Hence the cost of processing Q if X_i is substituted is equal to

$$(\text{cardinality of } R_i) \times \text{cost of processing } Q_i. \tag{5.1}$$

Therefore the first thought is to choose X_i with the smallest range. However this is not optimal for several reasons.

First, it may be possible to reduce some or all of the relations $R_1, R_2, \ldots, R_n$, by preprocessing one-variable clauses. Should this be done for all, for some, or for none of the variables? If all of the R_i can be reduced, this decision alone involves 2^n choices. The situation is further complicated by the fact that for a given variable the decision as to whether to preprocess the one-variable clauses depends on whether the variable is chosen for substitution. If it is to be chosen for substitution, then its range should be reduced as much as possible. If not, preprocessing may be a waste of time. On the other hand, which variable should be chosen depends not so much on R_i as on the reduced R_i. Let $Q(X_i)$ denote the one-variable subquery of Q in X_i and let R_i' be the reduced range after $Q(X_i)$ is processed. The following policies seem to be reasonable alternatives:

(a) Preprocess every $Q(X_i)$, basing the policy on the argument that the cost of processing one-variable queries is relatively small and it is important to choose the variable for substitution well.

(b) On the basis of $Q(X_i)$, decide for each variable whether to preprocess or not. Variable selection takes place after preprocessing.

The version of INGRES completed in January 1976 uses policy (a), partly because the variable selection is then based solely on the cardinalities of the reduced ranges. It is important, therefore, for these cardinalities to be accurate.

For (b) a workable policy is to use $Q(X_i)$ to estimate the size of R_i' for each i, and preprocess only if X_i is likely to be a contender for selection. For example, we might choose the top three contenders for preprocessing, or preprocess every variable for which the estimated size of R_i' is less than $\min |R_j|$. One good feature of (b) is that except for very unusual situations, the actual variable selected will be among those which have been preprocessed, and no further processing is necessary before substitution.

A second and more important objection to the strategy of choosing X_i with the smallest range is that the complexity of Q_i can vary greatly with i, and this must be taken into account in any strategy which lays claim to being even near-optimal. What must be determined is the extent to which Q can be reduced as a consequence of substituting for X_i.

Assume that we choose either (a) or (b) for the policy on preprocessing one-variable clauses so that that decision is decoupled from the selection of variable. We can assume that the query at this point consists of a single irreducible component with some one-variable clauses. The crux of the matter is how the irreducible component is affected by the substitution. Assume that whatever preprocessing is to be done has been done. Let the query be denoted by Q. Let $X_1, X_2, \ldots, X_n$ be the variables, and let $R_1, R_2, \ldots, R_n$ be their ranges. Let $Q_i(\alpha)$ denote the resulting query from substituting α for X_i in Q. Let C(Q) denote the *minimum cost* of processing Q. Then

$$C(Q) = \min_i \left\{ \sum_{\bar{R}_i} C(Q_i(\alpha)) \right\}, \tag{5.2}$$

where $\bar{R}_i$ denotes the set of tuple values which have to be substituted for X_i. In most instances this is simply R_i, although as we indicated earlier there are exceptions.

Equation (5.2) is a dynamic programming equation for the optimization problem at hand. As it stands, it is not too useful, since how C(Q) depends on Q is not known. However, (5.2) is a suitable starting point for optimization. The variable selected will correspond to the value of i which minimizes an *estimated value* for

$$C_i = \sum_{\alpha \in \bar{R}_i} C(Q_i(\alpha)). \tag{5.3}$$

Although we have in effect transferred the optimization problem to one of estimating cost, the latter is amenable to a variety of heuristic approaches. Consider some of these:

(a) Suppose we take the estimate of $C(Q_i(\alpha))$ to be independent of α and i. Then the minimum C_i corresponds to the smallest R_i. This somewhat simplistic policy is what has been implemented in the version of INGRES operational as of January 1976.

(b) We observe that unlike Q, $Q_i(\alpha)$ is not irreducible. One should therefore call reduction-subquery-sequencing to reduce $Q_i(\alpha)$ to a sequence S_i of subqueries, each of which is irreducible except for one-variable clauses. Now, α enters the subqueries only as a parameter, and the sequence S_i is really independent of α. Thus we have

$$C(Q_i(\alpha)) = \sum_{q_\alpha \in S_i} C(q_\alpha). \tag{5.4}$$

Since the structure of $Q_i(\alpha)$ has now been represented, we can accept a relatively crude estimate for $C(q_\alpha)$. For example, we might take the estimate of $C(q_\alpha)$ to be

$$C(q_\alpha) = \prod_j P(R_j), \tag{5.5}$$

where R_j are the ranges of q and P(R) is the number of pages occupied by R.

(c) We might try to obtain an estimate for cost by sampling. Consider the equation obtained from using (5.2):

$$C(Q) = \min_i \left\{ \sum_{\alpha \in \bar{R}_i} \sum_{q \in S_i} C(q_\alpha) \right\}. \tag{5.6}$$

This is truly recursive, since Q and q_α are queries of the same restricted type (viz. irreducible except for one-variable clauses). If the number of variables in Q is not enormous (in practice very few queries contain more than four or five variables), one might try to push the recursion (5.6) all the way down to one-variable queries, but using small samples for the range relations of Q. It is very likely that the costs of different paths in the decision tree vary widely, and only a few are contenders for the optimal path. With efficient management, this approach need not be prohibitively expensive.

These are but three possible approaches to estimating C(Q). Other approaches including some variants and combinations of these are under consideration. We expect to implement at least the three outlined above for experimental evaluation. Indeed, (a) has been implemented, and (b) is in the process of being implemented.

In order to use (5.5) in (5.4), we must know the number of pages oucupied by the range relations for every q in the sequence S_i. We note that S_i is a sequence and not a set, so that the range relation of a query may involve the result relations of queries which precede it. Therefore knowing the sizes of the range relations of Q is not sufficient to determine (5.5) for the q_α. Since we don't want to execute the sequence S_i except for the optimal i, we must rely on a procedure to estimate the sizes and other parameters of the result relation for a query.

Consider a query Q with range relations $R_1, R_2, \ldots, R_n$, a target list T(X), and a qualification B(X). Let the domains of R_i be denoted by D_{ij}, $j = 1, 2, \ldots, d_i$. Each R_i is by definition a subset of $\prod_{j \leq d_i} D_{ij}$. Hence, the product $\prod R_i$ is a subset of

$$D = \prod_{i \leq n} \prod_{j \leq d_i} D_{ij}. \tag{5.7}$$

To determine what subset of $\prod R_i$ satisfies $B(X) = 1$ requires accesses to the actual relations, but to determine what subset of D satisfies $B(X) = 1$ only requires knowing the domains $\{D_{ij}\}$. The storage required to represent $\{D_{ij}\}$ is in general far less than that required for $\{R_i\}$.

Let R(Q) denote the result relation of Q. We can estimate the cardinality of R(Q) as:

$$|R(Q)| = |\prod_{i \leq n} R_i| \cdot \{\text{fraction of D satisfying } B(X) = 1\}. \tag{5.8}$$

The domains of R(Q) can be estimated by evaluating T(X) on the subset of D which satisfied $B(X) = 1$. That is, the k-th domain of R(Q) is estimated to be

$$\{T_k(X); X \in D, B(X) = 1\}. \tag{5.9}$$

In most cases D_{ij} has sufficient regularity to permit it to be represented by just a few parameters. For example, D_{ij} might be simply all integers between a and b. Thus the storage requirement for keeping track of the domains for the result relations of the sequence S_i can be expected to be reasonable.

Since the sizes of the tuples are always known, the number of pages required for each of the result relations for the sequence can be computed from the estimated (5.8), which in turn is computed from the estimated domains using (5.9).

6. SUMMARY

In this paper we have presented a detailed account of how multivariable queries are decomposed in system INGRES. The basic ingredients of the decomposition are two: (a) to discover pieces of a query which are joined to the remainder by a single joining variable, and (b) to substitute for a variable. The overall strategy is to break up a query at the joining variables whenever this is possible, and to select a variable for substitution which incurs a "minimum cost" whenever substitution can no longer be postponed. A detailed algorithm for reducing a query into irreducible components has been given. Alternative approaches to estimating costs have also been discussed.

Optimization itself incurs a cost which has not been taken into consideration. For simple queries elaborate optimization may well do more harm than good. We have chosen an approach suggested by Stonebraker to resolve this. Suppose that we have two or more strategies st_0, st_1, . . . , st_n, each one being better than the previous one but also requiring a greater overhead. Suppose we begin a query on st_0, and run it for an amount of time equal to a fraction of the estimated overhead of st_1. At the end of that time, by simply counting the number of tuples of the first substitution variable which have already been processed, we can get an estimate for the total processing time using st_0. If this is significantly greater than the overhead of st_1, then we switch to st_1. Otherwise we stay and complete processing the query using st_0. Obviously the procedure can be repeated on st_1, to call st_2 if necessary, etc. For example, st_n may correspond to progressively more levels in the decision tree, or to progressively more elaborate estimates of result parameters, or to better sampling.

We have not addressed the question of optimizing the processing of one-variable queries. Some optimization is currently being done in INGRES, and this is described elsewhere [13].

In Appendix A we give a brief description of how INGRES is implemented. The original design of the implementation was primarily the work of Stonebraker and Held. Redesign of the third process and in particular the design of the query tree and the implementation of the decomposition algorithm in the current version (as of January 1976) have been largely the work of Peter Kreps. In Appendix B are specifications for the principal data structures needed for our decomposition algorithm.

One of us (E.W.) is responsible for introducing the conceptual framework in which the decomposition algorithm rests, viz. the policy of transforming a multi-variable query to one-dimensional ones, and the strategy of alternating between reduction and tuple substitution. We have collaborated on the reduction algorithm, and on the heuristics for variable selection. The implementation of the full algorithm as well as monitoring subsystems for the performance evaluation is being designed and executed by K.A.Y. The decomposition algorithm, being at the heart of INGRES, has enjoyed the attention of many participants of the project. It is difficult to remember who suggested what, but the three aforementioned colleagues have all made important contributions. In particular, as in every aspect of INGRES, the influence of Stonebraker is discernible throughout our algorithm.

APPENDIX A. SYSTEM ORGANIZATION

INGRES, Interactive Graphics and Retrieval System, runs on a PDP 11/45 under the UNIX operating system [8]. The entire system is written in the programming language C [9]. It has four major components, which are organized as shown in Figure 5. These four components are set up as processes under UNIX and communicate through the use of pipes. The user interface can be one of several forms: an in-

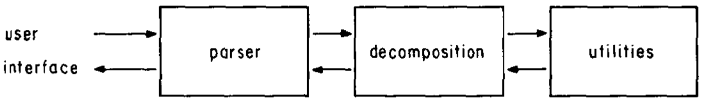


Fig. 5. Process configuration

teractive text editor, a graphics interface [5], an interactive English-like language [3], or part of a host programming language [1]. The parser accepts the user's query and processes it into a tree in conjunctive normal form. This query tree and a table of relations declared in the RANGE statements are passed to decomposition. The decomposition process contains not only the decomposition algorithm but also the one-variable query processor. The utilities process contains many functions which can be used by the system or the user.

APPENDIX B. DATA STRUCTURES

There are three main data structures which are used during decomposition of a query.

Range table. Some of the information for this structure is gathered during parsing and passed to decomposition as an ordered matrix. It is then put into a matrix, each entry of which has the following form:

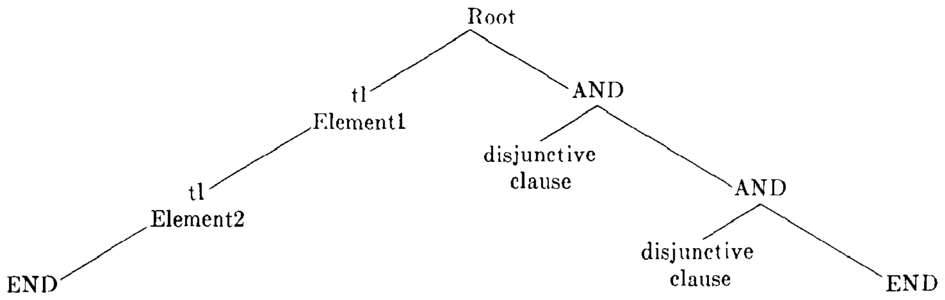
```
struct  rangev
{       char relid [MAXNAME];
        struct descriptor *desp;
        int setup;
}
```

The parser sends a table of relation names which have been declared in RANGE statements; the order of these names indicates the variable associated with each. These are relid. The second entry is a pointer to an in-core copy of the system catalogue description for that relation. The third entry is a flag which is set when the corresponding variable has been selected for substitution.

The use of this table aids decomposition in the use of temporary relations. When a new range is created for a variable by execution of a one-variable query, the entry in the range table for that entry is the same except for the pointer to the catalogue description. The relid is always the original relation name for that variable and the descriptor is for the current subrelation it is ranging over. In this way if a temporary relation must be created several times during the process of substitution, the same temporary relation name and descriptor can be reused by simply deleting the old tuples from the previous iteration. This saves much overhead in the creation of temporary relations.

Incidence matrix. This is a binary matrix of clauses (or subqueries) versus variables which is used within decomposition to represent the query currently under consideration. It is used during reduction to determine all irreducible subqueries and can be used during selection to represent the component subqueries in a compact form. This matrix also contains an entry for each clause which points to the actual clause so that it may be easily obtained when it is necessary to build a query tree for execution of a subquery.

Query tree. The parser sends a list representing the query tree to decomposition, which then rebuilds the query tree adding useful information as it is recognized. The general form of this tree is a root node with the target list of the query as the left branch and the qualification as the right branch. Since the query is in conjunctive normal form, all the intermediate nodes along the right side will be AND (conjunction) nodes.



More specifically, nodes of the tree are defined as:

```
struct querytree
{       struct querytree *left, *right;
        struct symbol sym;
}
```

where left and right are the pointers to the respective branches. The second entry defines the structure within the node and this varies depending on the type of node.

For nodes representing arithmetic operators, disjunctions (OR), result domains, and constants, the structure is:

```
struct symbol
{       char type;
        char len;
        int value[ ];
}
```

where type is a code representing the type of the node (i.e. plus, minus, OR, etc.) and len is the length in bytes of value. Value is a variable length field (0–255 bytes) and contains the appropriate value for that type of node. For example, if the node is representing a constant then the value contains the actual constant. For nodes representing variable.attribute (e.g. E.SALARY) the structure is:

```
struct symbol
{       char type;
        char len;
        char varno, attno;
        char frmt, frml;
        char *valptr;
}
```

where type is the same as above and len is fixed. varno is an index into the range table for the correct variable; attno is the domain number (from the system catalogue) of the correct domain referenced; and frmt and frml give the format of the attribute (e.g. A6, I2, etc.). This is used to determine new domain types and for calculations. The last entry is used during tuple substitution. If a particular variable is selected for substitution, all variable.attribute nodes involving that variable will become nodes representing constants. But the tree itself need not be changed. This field, valptr, is simply set to point to the constant value that should be used. This position remains fixed so when a new tuple value is substituted the pointer does not change; only the value it is pointing to changes. In this way a new tree is not needed for each level of substitution or for each iteration of substitution values. If the pointer is zero, the variable information is used; if it is nonzero, it is a constant node.

For nodes representing the root or conjunctions (AND), the structure is:

```
struct symbol
{       char type;
        char len;
        char tvarc;
        char lvarc;
        int lvarm;
        int rvarm;
}
```

where type is the same and len is fixed. tvarc and lvarc are both counts of the variables used; tvarc is the number of variables in the subtree below this node; and lvarc is the number of variables in the left branch. So for the root node, tvarc is the total number of variables in the query and lvarc is the number of variables in the target list. For an AND node tvarc is the number of variables in the remaining clauses and lvarc is the number of variables in the single clause of its left branch. lvarm and rvarm are bit maps of the variables used in the left and right branches of the node, respectively.

This structure is not as costly as it might appear. It is true that during decomposition many subqueries are created and executed many times, but it should be noted that all of these subqueries use clauses which appear in the original query. The target lists may change, but no new clauses are ever created except through substitution. Since this is true, when a subquery is to be executed a query tree can be constructed using nodes from the original tree. A new root node must be created for each subquery and for some target list nodes, but all the AND nodes can simply be detached from the original query tree and added to the new query tree.

REFERENCES

1. Allman, E., and Stonebraker, M. Embedding a relational data sub-language in a general purpose programming language. ERL Mem. No. M564, Electronics Research Lab., U. of California, Berkeley, Calif., Oct. 1974.
2. Astrahan, M.M., and Chamberlin, D.D. Implementation of a structured English query language. *Comm. ACM 18*, 10 (Oct. 1975), 580–588.
3. Codd, E.F. Seven steps to rendezvous with the casual user. Proc. IFIP TC-2 Working Conf. on Data Base Management Systems, Cargese, Corsica, April 1974.
4. Held, G.D., Stonebraker, M., and Wong, E. INGRES—a relational data base management system. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., pp. 409–416.
5. McDonald, N., and Stonebraker, M. Cupid—the friendly query language. ERL Mem. No. M487, Electronics Research Lab., U. of California, Berkeley, Calif., Oct. 1974.
6. Palermo, E.P. A data base search problem. Proc. 4th Int. Symp. on Computers and Information Science, Miami Beach, Fla., Dec. 1972.
7. Pecherer, R.M. Efficient evaluation of expressions in a relational algebra. Proc. ACM Pacific 75 Conf., April 1975, pp. 44–49.
8. Ritchie, D., and Thompson, K. The UNIX time sharing system. *Comm. ACM 17*, 7 (July 1974), 365–375.
9. Ritchie, D.M. C Reference Manual. UNIX Programmer's Manual, Bell Telephone Labs, Murray Hill, N.J., July 1974.

10. Rothnie, J.B. An approach to implementing a relational data base management system. Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.
11. Rothnie, J.B. Evaluating inter-entry retrieval expressions in a relational data base management system. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., pp. 417–423.
12. Smith, J.M., and Chang, P.Y.T. Optimizing the performance of a relational algebra database interface. *Comm. ACM 18*, 10 (Oct. 1975), 568–579.
13. Stonebraker, M., Wong, E., Kreps, P., and Held, G. The design and implementation of INGRES. *ACM Trans. on Database Systems 1*, 3 (Sept. 1976), 189-222 (this issue).
14. Todd, S. PRTV: An efficient implementation for large relational data bases. Proc. Int. Conf. on Very Large Data Bases, Framingham, Mass., Sept. 1975, pp. 554–556. (Available from ACM, New York).
15. Tsichritzis, D. A network framework for relational implementation. Rep. CSRG-51, Computer Systems Research Group, U. of Toronto, Toronto, Ont., Canada, Feb. 1975.

Received December 1975; revised April 1976