P. A. V. Hall

Optimization of Single Expressions in a Relational Data Base System

Abstract: This paper examines optimization within a relational data base system. It considers the optimization of a single query defined by an expression of the relational algebra. The expression is transformed into an equivalent expression or sequence of expressions that cost less to evaluate. Alternative transformations, and combinations of several transformations, are analyzed. Measurements on an experimental data base showed improvements, especially in cases where the original expression would be impracticably slow in its execution. A small overhead was incurred, which would be negligible for large data bases.

Introduction

Relational data base systems provide the user with a tabular view of the data, a view that is independent of any machine or implementation. The user need know nothing of the implementation in formulating his query. Thus he cannot be held responsible for the efficiency with which his query is answered.

Complex queries on large volumes of data take a long time to answer. But the speed of reply should not depend critically on the way the query or other request for processing is formulated. The intention of the user should be preserved, but the details of the query should be transformed to ensure a faster response; casual users would thus be protected from catastrophicly expensive queries. At the same time, an experienced user, who may well express his query concisely and not require optimizations, should not be penalized by an unnecessary overhead. However, even the experienced user may sometimes express complex queries inappropriately and thus benefit from such optimization.

This paper addresses the optimization of single queries to a relational data base. A query is written as a relational expression, which is transformed into an equivalent expression (or sequence of expressions). In this paper we explore the variety of transformations that are practicable. We then present the results of an experimental implementation of the transformations designed to investigate whether the transformations can be achieved with negligible overhead and whether they can successfully catch the unfortunately formulated query and transform it into one that can be answered effectively. It will be seen that in both cases the answer is affirmative.

The investigation into optimization reported here has been specifically aimed at an experimental prototype relational data base system [1]. We investigated only one facet of the optimization problem. There are longer term optimization methods, such as repositioning data on disk [2–4], retaining data once it has been computed in case it is required again [5], etc.

The optimization methods considered here are necessarily biased towards the particular system that we were attempting to optimize. A first essential is to arrange that all relational operations occur as efficiently as possible, using suitable storage strategies for relations (sorting, indexing, encoding, etc.). Whereas most of the optimization methods to be discussed are not dependent upon implementation details, some knowledge of the PRTV system implementation is necessary. (Additional information about PRTV is provided in [1]).

The PRTV system uses two levels of language. The external interface language (ISBL) is a relational algebra that uses symbolic names for relations and for components of tuples. At a lower level there is an internal language (CIL) in which relations are referred to by position. Various implementation aspects intrude at the CIL level, because the positions of components determine a sort order. In the PRTV implementation relations are stored as indexed sequential files, and operations are performed where possible by merging these files, exploiting the internally known sort orders. The files are sorted on column 1, then on column 2 for identical values in column 1, and so on. Projections not only reduce the number of components; where components are re-ordered, a sort operation is induced. Complex expressions are evaluated a tuple at a time, with all the merges performed together. Intermediate results are only formed when essential—for a sorting projection or at the second argument to a join.

The optimization methods considered in this paper are transformations of algebraic expressions in CIL. These expressions can, in fact, be very large, much larger than any expression that a user would write. This comes about because of another efficiency feature of the PRTV system, deferred execution. When an ISBL expression is written and the result assigned to some variable, the expression is not evaluated but merely "semicompiled" into a CIL expression, with arguments in ISBL being replaced by their corresponding CIL expressions. CIL expressions are only evaluated when it becomes essential to do so—for example, when the result is to be displayed for a user.

In this paper we write expressions using a self-explanatory notation built on the operations:

$A \cup B$—set union of relations A and B of the same type.
$A - B$—set difference of relations A and B of the same type.
$A \cap B$—set intersection of relations A and B of the same type.
$A \,\%\, T$—projection of relation A onto the components given in projection list $T = C_{i_1}, \cdots, C_{i_m}$.
$A : F$—selection of a subset of relation A of tuples for which filter F is true, where F is a logical expression involving the components of the tuples of A.
$A * B$—Cartesian product or "join" of relations A and B.

The sequence of a Cartesian product followed by a selection is a common compound operation recognized within the implementation and executed as a single operation. The above notation is not that used for CIL (CIL is an unreadable prefix polish) but is equivalent to it.

The transformations of expressions to be considered are:

1. Change the order in which the operations are performed while ensuring that the correct result is obtained. For example, make selections occur as early as possible.
2. Recognize common subexpressions and evaluate some of them separately first.

The legitimate transformations for use in 1) can be expressed using algebraic laws that show which expressions are equivalent to one another. For 2) we must be able to recognize when sequences of expressions and assignments are equivalent. Then our problem becomes that of selecting, from among all those expressions equivalent to the original, that expression or sequence of expressions that is the cheapest to evaluate.

A basic requirement is thus to be able to measure the cost of evaluating a given expression or sequence of expressions. This is studied in the Appendix, where we find that estimating the cost of evaluating expressions is extremely difficult; instead of attempting to find the best possible expression to evaluate, we are forced into weaker, heuristic procedures for finding expressions that we might reasonably expect to be better.

These ameliorating transformations are discussed in successive sections. First we cover transformations that produce only a single expression from a given single expression, using algebraic transformations. We also discuss the detection of common subexpressions and the replacement of a single expression by a sequence of expressions. Next we discuss the various forms of optimizer used in the PRTV prototype and present the results of an initial evaluation of three different levels of optimizer.

Ameliorating and optimizing transformations

Because of the difficulty of estimating evaluation cost, it seems best to avoid any transformations that depend critically upon cost. Thus our approach has been to make ad hoc transformations that are generally felt to be useful, as well as applying those few rules that are always guaranteed to improve performance.

In discussing the various transformations, it is both conceptually and analytically useful to view expressions as trees in which the leaves are the stored relations, filters, or projection lists, whereas the internal nodes are the operators. A first simple example appears in Fig. 1. Later we view the existence of common subexpressions in terms of lattices formed from the expression trees.

We consider ameliorating transformations in two stages. First we examine transformations of a single expression that retain the single expression, and then consider transformations that break up the expression into a sequence of expressions. Finally we consider the order in which to apply the transformations.

• *Transformations within a single expression*

The simplest transformations that are generally applicable are the combination of sequences of projections into a single projection and the combination of sequences of selections into a single selection.

The more complex generally applicable transformations involve the removal of null relations and redundant operations, such as $A - A$ and $A \cup A$. At first sight this appears trivial, but in fact it rests on the identification of common subexpressions, followed by the application of various algebraic laws that remove the redundancies. Later we discuss these laws, the identification of common subexpressions, and how to combine these operations.

Only one heuristic transformation is considered. All selections must be moved so that they are performed as early as possible; that is, they are moved down the

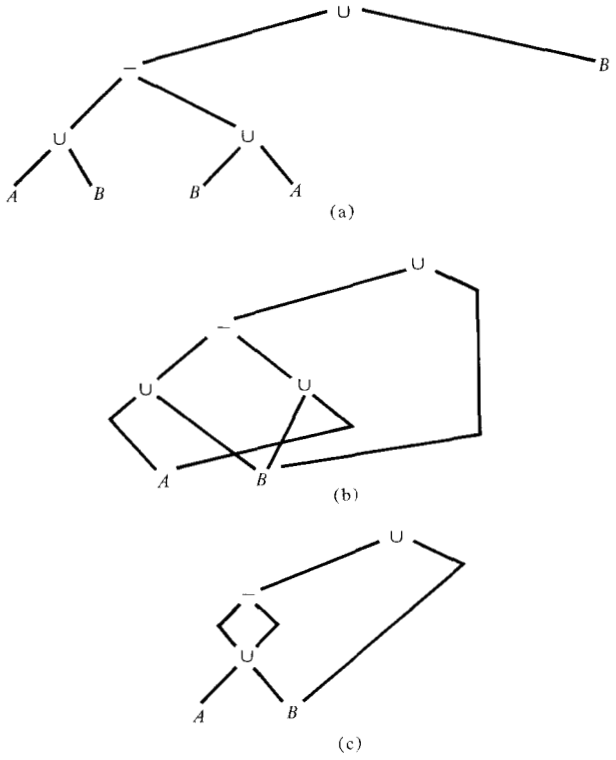

Figure 1 (a) Expression tree of [(A ∪ B) − (B ∪ A)] ∪ B; (b) intermediate step in converting to lattice; and (c) lattice resulting from removal of common subexpressions.

expression tree, as far as the leaves if possible. The justification is that selections usually reduce cardinality significantly, but we will see examples where these transformations do not help. Some of the manipulations involved in moving selections are quite complex (notably, where joins are involved). The movement of selections is discussed later.

Combining sequences of projections and selections

A sequence of selections can very simply be replaced by a single selection with a larger filter. The new filter is simply the conjunction of the original filters. The whole transformation can be captured in the formula

$$\{\cdots[A : F_1) : F_2]\cdots F_n\} = A : (F_1 \,\&\, F_2 \,\&\cdots\&\, F_n).$$

We are assuming that it is cheaper to test a tuple of relation A using the combined filter than to test it first against F_1, then against F_2, and so on. It is cheaper for the implementation of PRTV but could conceivably not be for other implementations. The collection of all the filters has other advantages when moving filters down the expression tree, as will be seen.

A similar combination of sequences of projections into a single projection is possible:

$$A \,\%\, T_1 \,\%\, T_2 \,\%\cdots\%\, T_n = A \,\%\, T,$$

where the T, $T_1, \cdots, T_n$ are projection lists. List T is formed from $T_1, \cdots, T_n$ using the following algorithm. Each projection list has the form

$$T_k = C_{i_{k1}}, C_{i_{k2}}, \cdots, C_{i_{kn_k}},$$

where i_{kj} is an integer, being the jth component number of the T_k list.

For example, consider the expression

$$A \,\%\, C_3, C_1, C_2 \,\%\, C_1, C_3 \,\%\, C_2.$$

Tracing through the expression we see that the last projection, $()\%C_2$, requires that we place in position 1 of the result the value that was in position 2 previously; from the previous projection, $()\%C_1, C_3$, we see that this value came from position 3 earlier. Thus the expression reduces to

$$A \,\%\, C_3, C_1, C_2 \,\%\, C_3.$$

Working backwards another step, the expression finally reduces to

$$A \,\%\, C_2.$$

The general algorithm works in exactly this way. It combines T_n and T_{n-1} into a single projection list, calling this T_{n-1}; then it combines this with T_{n-2}, and so on, until a single projection list is left. The combination of projection lists is essentially a composition of functions.

Algorithm1 – projection sequence reduction

Projection sequences and lists are as defined above.

```
DO k FROM n TO 2 BY −1;
    DO j FROM 1 TO n_k;
        In T_k replace C_{i_kj} by C_{i_(k−1)i_kj} from T_{k−1};
    END;
    Replace T_{k−1} by T_k;
    k ← k − 1;
END;
```

The saving from combining sequences of projections could be dramatic. Storing all relations in sort order means that projections could cause the complete resorting of a relation. A sequence of n projections could involve as many as n sort operations, which is reduced to at most one sort by the reduction of the sequence to a single projection. We could even reduce n sorts to no sorts, if the final projection fortuitously led to a relation in the same sort order as the original.

Idempotency and null relations

Any relational operation involving a null relation ∅ as one of its arguments is a redundant operation and can be removed. The appropriate reductions for each operation are shown in Table 1. Note that these laws would need application from the bottom of a large expression upwards, so that expressions like

$(A \cap \varnothing) \cup A$

would get reduced as far as possible. This bottom up reduction is expressible recursively, by first using a recursive invocation to remove the null relations from the subexpressions that are the arguments to the top operation, followed by the application of the reductions of Table 1.

Some relational operations not involving null relations, but involving the same argument repeated, are redundant and can be eliminated. These laws are known as idempotency laws and are shown in Table 2. Notice that for differences we obtain a null relation as the result, so the idempotency laws should be applied concurrently with the removal of the null relations.

However, the application of idempotency is not so simple, because we could encounter idempotency at any level in the expression due to the presence of common subexpressions. Thus

$$[(A \cup B) - (B \cup A)] \cup B$$

should be reduced to a B, which means recognizing the common subexpression $(A \cup B)$. Thus an integral part of the application of the idempotency laws must be the recognition of common subexpressions. It is not enough to recognize common subexpressions first and then apply idempotency and null relation removal, because we would fail to reduce

$$\{(A \cup B) - [(B \cup \varnothing) \cup A]\} \cup B.$$

Thus common subexpression recognition and idempotency and null relation removal must be performed concurrently. Moreover, we need to extend our processes for common subexpression recognition to all parts of the relational expression, in particular to filters, which are Boolean expressions with their own idempotency and null expression laws, as shown in Table 3.

How do we recognize common subexpressions? Solutions to this have been explored in depth in an earlier paper [6], but the method is sketched here for completeness. The point of departure in the method is to notice that expression trees in which a common subexpression occurs really have a lattice structure, in which edges converge onto common subexpressions. Figure 1(a) shows an expression tree, whereas Fig.1(c) shows this tree redrawn as a lattice to emphasize the common subexpression. The process of analyzing an expression for common subexpressions is equivalent to converting from a tree as in Fig.1(a) to a lattice in which no nodes are redundant, as in Fig. 1(c). This conversion can best be achieved by working from the bottom level of the tree, the leaves, upwards a level at a time, introducing all convergences possible at each level before moving up to the next level. Thus, in automatically converting the tree of Fig. 1(a) into the lattice of Fig. 1(c), we first note that leaf A appears twice and remove one, and note that leaf B appears three times, and remove two of these, the result having considerable convergence, as shown in Fig. 1(b). Then stepping up a level from the leaves, we find that of the three Us only two should be considered because these two have both arguments in the previous level. These two Us have the same arguments and give the same result when we take into account that the union operation is commutative. Thus one of these is redundant and can be removed, leading to the lattice of Fig. 1(c).

The final algorithm is then as shown below.

Table 1 Reduction of relational expressions involving null relations.

Expression	*Reduces to*
$\varnothing \cup A \equiv A \cup \varnothing$	A
$\varnothing \cap A \equiv A \cap \varnothing$	$\varnothing$
$A - \varnothing$	A
$\varnothing - A$	$\varnothing$
$\varnothing \% T$	$\varnothing$
$\varnothing$: F	$\varnothing$
A : true	A
A : false	$\varnothing$
$A * \varnothing \quad \varnothing * A$	$\varnothing$

Table 2 Idempotency laws of relational algebra.

Expression	*Reduces to*
$A \cup A$	A
$A \cap A$	A
$A - A$	ϕ

Table 3 Null expression and idempotency laws of Boolean algebra.

Expression	*Reduces to*
A & true = true & A	A
A & false = false & A	false
$A \vee$ true = true $\vee A$	true
$A \vee$ false = false $\vee A$	A
A & A	A
$A \vee A$	A
A & $\neg A = \neg A$ & A	false
$A \vee \neg A = \neg A \vee A$	true

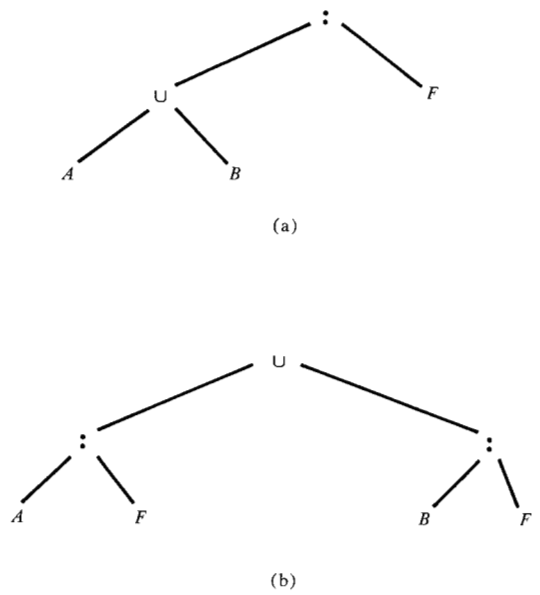

Figure 2 Distributing a filter through a union, viewed as "moving the filter down the expression tree." Expression (A ∪ B) : F is shown at (a) before distribution; at (b), after distribution, we have (A : F) ∪ (B : F).

Algorithm 2—common subexpression identification, idempotency application, and null relation removal

Start at the leaves of the tree, and work up the tree a level at a time, removing any repeated common subexpressions and applying the various algebraic laws of Tables 1, 2, and 3, until no further reductions in the expression occur. For a full account of this, see [6].

Performing selections as early as possible

We try, if possible, to transform the expression so that all selections are performed as early as possible, moving them so that they are directly associated with primitive stored relations. Thus, for example, if a union is followed by a selection, we would distribute the selection through the union, to obtain the following transformation:

$$(A \cup B) : F = (A : F) \cup (B : F),$$

where A and B are relations, $\cup$ means union, and F is a filter or selection logical expression. This can be viewed graphically as moving the filter down through the union, as illustrated in Fig. 2.

For each operation there is a similar distribution law. All the laws are shown in Table 4. To transform an expression, we would then recursively apply the appropriate transformation as set out in Table 4, pushing the filters down the tree as far as they will go. They will move down as far as the leaves unless they get stuck at a join. This is embodied in the following algorithm.

Algorithm 3—moving filters down expression trees

```
TRANSFORM: PROCEDURE(EXPRESSION)
                                    RECURSIVE;
  IF EXPRESSION is atomic THEN RETURN;
  IF EXPRESSION of form SUBEXP; FILTER
THEN distribute FILTER into SUBEXP using
       Table 1 to form expression (SUBEXP1 OP
       SUBEXP2); FILTER RESIDUE;
  ELSE EXPRESSION already of form SUBEXP1
                                    OP SUBEXP2;
  TRANSFORM(SUBEXP1); TRANSFORM
                                    (SUBEXP2);
```

The distribution laws in Table 4 are obvious for union, intersection, and difference. Some alternatives are given that require less filtering, but these are not preferred for two reasons. First, the reason for the transformations is that filters reduce cardinality significantly and so should be worth the repetition. Second, by making a symmetric distribution we do not destroy common subexpressions that lead to a successful application of the idempotency laws discussed in the last section. The combination of pairs of selections into a single selection was discussed but has been included in Table 4 for completeness.

To change the order of filters and projections requires modifying the filter so that the references to components of tuples within the filter remain correct. Thus if the projection places the ith component of tuples of relation A into the jth position of tuples of $A \% T$ (i.e., projection tuple T has an i in its jth position), then every reference in filter F to component j must be changed to a reference to component i in filter F'.

The only really complex transformation occurs for joins. In general the filter associated with a join refers to both of the relations being joined. However, there may be parts of it that refer only to one or the other of the argument relations, and this part could possibly be factored out and moved down through the join. Thus we wish to transform filter F into an equivalent F_1 & F_2 & F_3, where filter F_1 refers only to components of relation A, filter F_2 refers only to components of relation B, whereas filter F_3 refers to both relations A and B. Clearly F_1 and F_2 should be as large as possible. A detailed analysis of how this can be done is given in [7], where a couple of examples are given to show how the method works. Filters F_1 and F_2 can be moved down through the join, leaving behind filter F_3, which contains only the essential parts of the filter that select on information jointly from both arguments to the join. The combined operation $A' * B' : F_3$ is performed as a single operation and, if F_3 is in the appropriate form, this operation can be performed very efficiently. Thus F_3 should be further analyzed and transformed to ensure an efficient join.

Table 4 Distribution laws for moving filters down expression trees.

Operation	*Expression*	*Transformed to*	*Alternatives*
selection :	$(A : F) : G$	$A : (F \,\&\, G)$	
union $\cup$	$(A \cup B) : F$	$(A : F) \cup (B : F)$	
intersection $\cap$	$(A \cap B) : F$	$(A : F) \cap (B : F)$	$(A : F) \cap B, A \cap (B : F)$
difference $-$	$(A - B) : F$	$(A : F) - (B : F)$	$(A : F) - B$
projection %	$(A \,\%\, T) : F$	$(A : F') \,\%\, T$ (F' is derived from F; see text.)	
join *	$(A * B) : F$	$((A : F_1) * (B : F_2)) : F_3$ where $F = F_1 \,\&\, F_2 \,\&\, F_3$ (See text).	

In general the join operation has to be performed as a full Cartesian product (the so-called quadratic join) to produce $n(A')n(B')$ tuples from which the relevant tuples are selected. In our experimental system the relations are stored in sorted order and, if the filter F_3 is "nice," the whole join can be executed as a merge process with no more effort than required for an intersection. In fact, intersection can be viewed as an extreme best case of join in which the filter belonging to the join equates component 1 of A' with component 1 of B', component i of A' with component i of B', for all i. If only the leading few components are equated, we still can do the join almost as efficiently. Thus filter F_3 should be re-expressed in the form

$$F_3 = C_1 = C_j \,\&\, C_2 = C_{j+1} \,\&\, \cdots \,\&\, C_k = C_{j+k-1} \,\&\, C_{k+1} \,\mu\, C_{j+k} \,\&\, F_4$$

for as large a k as possible. The μ is any comparison operator other than $=$. More complicated arrangements than this would be advantageous but were thought to be so unlikely that they were not worth the computational overhead necessary to detect them. The method for achieving the above transformation is exactly the same as that used for performing the factorization of F into F_1 & F_2 & F_3 described in [7].

Now transformations like the distribution of a filter into a union do not necessarily improve things. If A and B are disjoint, then the cardinality of their union is the sum of their individual cardinalities and filter F is applied as many times both with and without the transformation. Thus applying the distribution must always be favorable. But if A and B overlap, then for some tuples, if the filter is distributed, it will be applied twice, which may not compensate for the saving of work in the union obtained by performing the filtering first. In some cases, the filter may not change the cardinality much, and then we lose by moving the filter down the expression tree. For intersection and difference this worsening of the situation when the filter does not change the cardinality significantly is even more marked. However, for joins, especially where the join is necessarily a full quadratic join, the transformation almost always improves things, because that part of the filter moved through the join is executed $n(A)$ times rather than $n(A)\,n(B)$ times (or $n(B)$ times rather than $n(A)\,n(B)$ times). Note, however, that attaching filters to leaves has an important added advantage. During the evaluation, the relation at a leaf is stored, and the presence of a filter would enable the system to use any indexes or inversions present. This could lead to significant savings compared to the alternative of reading through all the tuples of the relation and selecting those desired using the filter. This latter course is the only course available for filters positioned at nodes other than leaves.

• *Transformations of a single expression into a sequence of expressions*

A subexpression that occurs in several different places within an expression need only be evaluated once. Evaluating a subexpression once, storing the result, and then using it many times has a cost overhead in storing and retrieving the intermediate result. This must be compared to the saving obtained by not evaluating the expression repeatedly. Not all common subexpressions are worth separate evaluation, and the decision to store the result of one subexpression affects the decisions concerning other subexpressions. We have a complicated discrete optimization problem.

The identification of common subexpressions is a necessary step. This would be achieved by the method given earlier and can be viewed as a by-product of Algorithm 2. Because, as we will see later, Algorithm 2 is

executed before any methods of this section, we do not consider the recognition of common subexpressions further, but assume that these have been located and represented by a lattice, as in Fig. 1(c).

We have two considerations to explore. How much do we save by evaluating a common subexpression only once? If we have many possible common subexpressions, which do we select?

Deciding which common subexpressions to eliminate

Can we decide when a common subexpression is worth eliminating and when it is not? For this decision we need to find a formula for the saving that might be obtained and satisfy ourselves that this formula is sufficiently robust given any uncertainties about its validity.

Suppose that a subexpression occurs M times within some large expression. Suppose that we have

C_e = cost of evaluating the subexpression only once,
C_s = cost of storing the result of the subexpression,
C_r = cost of retrieving the result of the expression once it has been evaluated and stored.

Our two choices are 1) Evaluate the subexpression each of the M times its value is required (which costs MC_e); 2) evaluate the subexpression once only, store the result, and then read it back each time it is required (which costs $C_e + C_s + MC_r$).

We decide upon choice 2) if it costs less; that is, if

$$MC_e > C_e + C_s + MC_r$$

or if

$$\Delta = (M-1)\,C_e - C_s - MC_r > 0.$$

The Δ is the discriminant function, which computes the saving obtained by choice 2), namely evaluating the common subexpression only once.

Decisions concerning multiple subexpressions

In general expressions contain many common subexpressions, and decisions about one common subexpression affect the decisions regarding the others. Thus, when considering a large expression containing many common subexpressions, we would not be able to examine each subexpression independently of any decisions made concerning the rest. We must consider all possible combinations and evaluate them as a whole. For N common subexpressions, we have 2^N possible selections, and to examine each of these exhaustively would be prohibitive. We need a more structured approach and must either select the best possible combination using some general purpose discrete optimization method, such as dynamic programming or branch-and-bound methods, or we must resort to fast methods that produce acceptable suboptimal results.

Because the general purpose exact approaches are complex to implement and often are inefficient in execution, heuristic methods were studied. The simplest ad hoc method that does not involve any form of blind guess is to "hill-climb" towards the "best." The idea is to select the best subexpression to be evaluated once and stored, and then to look for the next best subject to this first decision, and so on until no further improvements can be made. This method involves of order N^2 work. However, it can not be guaranteed to find the global optimum, although it will find a local optimum.

Working through several examples, it was always found that the hill-climb method leads to the global optimum. Thus this method has been implemented. In programming this method, various implementation conveniences have resulted in modification of the basic idea. This has led to the following algorithm, which evaluates a common subexpression as early as possible and uses the information gained to modify all the cost estimates for successive decisions.

Algorithm 4—common subexpression evaluation Let the lattice formed from the expression be L.

$L_1 \leftarrow L$;
$k \leftarrow 1$;
WHILE $k > 0$ DO;
  Let $N = \{n, \cdots, n_m\}$ be the nodes of L_k such that for each n_i, storing the intermediate result formed at n_i leads to an improvement in evaluation.
  CASE $N \neq \varnothing$;
  Select that n_i in N that yields maximum improvement.
    Let L_{k+1} be the lattice contained in L_k with n_i as supremum.
  $k \leftarrow k + 1$;
  CASE $N = \varnothing$
    Evaluate subexpression L_k. Use correct cardinality thus obtained to correct the current estimates of cardinality and costs in the remaining lattices.
  $k \leftarrow k - 1$;
END;

• *Order of transformations*

Table 5 summarizes the transformations that we have discussed. Ideally, the order in which the transformations are made should be unimportant. We would like a situation in which any order leads to the same result. Such a situation is known as Church-Rosser [8] but unfortunately does not apply here.

As we move filters down the expression tree, we can bring together projections previously separated by a selection. Hence transformations of Algorithm 3 should be done before those of Algorithm 1, although of course these could be done within the same recursive traversal of the expression tree.

If we start with an expression containing a repeated common subexpression and if we transform it using Algorithms1 and 3 before detecting the common subexpressions, etc., using Algorithm 2, we may have destroyed or at least radically reduced any common subexpressions. Consider the expressions

$$(A \,\%\, T \,\%\, S) + (A \,\%\, T)$$

$$(A \cup B) : F + (A \cup B) ; G.$$

In the first case, combining the projections loses the common subexpression $(A \,\%\, T)$, whereas in the second case, moving the filters loses the common subexpression $(A \cup B)$.

Fortunately the common subexpressions necessary for the application of idempotency and null relation removal are not affected if either Algorithms 1 or 3 are executed before Algorithm 2, providing only that the standard symmetrical transformations of Table 4 are used. Thus between Algorithms 1 and 2, and between Algorithms 3 and 2, we have a mutual Church-Rosser property, and the local order of application is unimportant. This is not proved, because it is felt to be obvious. It is important, however, for it influences our decisions concerning the final order in which we should apply our four transformation algorithms. The order of application of the algorithms considered thus far would be Algorithm 3, then Algorithm 1, then Algorithm 2.

Once we break up the large expression into a sequence of expressions as part of the common subexpression elimination of Algorithm 4, we find that we have a conflict. The movement of filters down the expression tree can produce enormous savings, yet this destroys many common subexpressions. Consider the expression

$$(BI : C_5 = \text{'computing'}) \cap (BI : C_5 = \text{'mathematics'}),$$

where the BI is itself a large expression. If we move the filters first, the work is dramatically reduced. But different filters move down the two branches, and the large common subexpression BI disappears, although there may still be a considerable residue in common.

In this example, it is clearly best to move the selections first, because these lead to a far more dramatic improvement than the common subexpression elimination. In general both possibilities as well as all intermediate possibilities would need individual consideration and costing. However, it was felt that this was not worth the effort involved, so a blanket decision was taken to move filters first, and only afterwards eliminate common subexpressions.

Thus the order of application of the various transformations that has been used in the PRTV relational data base system is:

Table 5 Summary of optimizing transformation algorithms.

Algorithm	*Transformations made*
1	Combine sequences of projections into a single projection.
2	Identify common subexpressions. Remove redundant relation operations (idempotency and identity removal). Remove redundant operations in filters.
3	Perform selections as early as possible. Combine sequences of selections into a single selection.
4	Common subexpression evaluation. Select those common subexpressions that are worth evaluating once only and storing with the result retrieved as required.

First Algorithm 3. Move selections down the expression tree so that they are performed as early as possible.

Second Algorithm 1. Combine sequences of projections.

Third Algorithm 2. Eliminate redundant operations, idempotency, and null relations.

Fourth Algorithm 4. Eliminate common subexpressions.

Note that once a transformation has been performed, it does not need to be repeated later. Our chosen order of application ensures this. In Algorithm 2, where a need to cycle during common subexpression detection and redundancy removal emerged, we avoided cycling by doing the transformations during the same pass through the expression.

Experimental validation of the "optimizing" transformations

The ameliorating transformations presented in the preceding sections have been implemented within the PRTV system. The implementation has proceeded in three phases, giving three levels of optimizer. These phases were dictated primarily by the degree of integration necessary with the rest of the system. In phase 1 no access to the data base is necessary; in phase 2 the data base must be accessed to obtain the degree of the relation, whereas in phase 3 cardinality must be obtained and intermediate results computed and stored.

- *Phase 1: Mini-optimizer*

This phase makes no access to the data base, transforming the expression purely at the syntactic level. It uses Algorithm 3 without the manipulation of filters at joins, and it uses Algorithm 1.

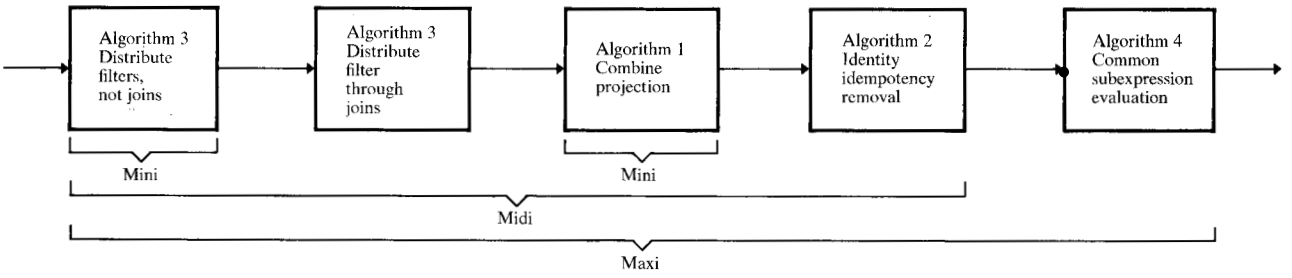

Figure 3 Relationship among the three levels of optimizer and the transformations.

• *Phase 2: Midi-optimizer*
This phase accesses the data base to obtain degrees of the relations. It uses all of Algorithm 3 and Algorithm 1 and Algorithm 2.

• *Phase 3: Maxi-optimizer*
This phase accesses the data base for degree and cardinality of the stored relations to evaluate intermediate results. All algorithms are used.

Figure 3 shows how the three levels of optimizer relate to the algorithms presented earlier. Note that each level includes all the transformations of the lower levels.

The implementation of the transformations was undertaken as part of the process of validation of the transformations. To obtain some feel for the gains that might be obtained by the transformations, we ran a series of tests using the three levels of optimizer. These tests are not intended to measure accurately the savings obtained by using the transformations but rather to verify that

1. Extreme cases that might arise in practice with the use of the system for complicated queries, or queries made by personnel unfamiliar with data processing, are caught by the transformations and radically improved.
2. Queries from users who would write them efficiently are not degraded.

The tests were also undertaken to pinpoint any problem areas where further transformations would be useful.

To evaluate completely the ameliorating transformations would require "typical" patterns of data base activity or alternatively the setting up of a trial application specifically for test purposes. Information obtained from traditional data processing systems regarding typical queries cannot be extrapolated to relational data base systems because the greater flexibility of the relational systems promotes more complex queries. No real experience of relational data base systems over a wide variety of applications has been obtained, and thus one is forced into evaluating the levels of the optimizer using small amounts of synthetic data.

For the test, a library problem was simulated. Relations *ACQ*, *BRW*, *DDC*, and *HIST* were modeled by generating random numerical data for them, using distributions judged reasonable. The details are shown in Table 6.

A set of queries was timed using these data. The queries were partly selected to stretch the facilities of the levels of optimizer and were partly drawn from the queries that could arise in practice. The times taken to access the data base and return the result without storing it were obtained without the optimizer and with the three levels of optimizer. CPU time was recorded; this fluctuated from trial to trial, and the smallest of a series of trials was taken.

Table 7 shows the various expressions that were used as queries, together with the transformed expressions actually evaluated in the three levels of optimizer.

The times obtained during the experiments are shown in Table 8. Let us work through these and account for the observed times.

Query 1 is *HIST*−*HIST*. The relation *HIST* is subtracted from itself. This kind of pathological query could arise in practice as a means of setting the contents of a relation to the empty set. Due to deferral of operations, in general the query would be of the form $X - X$, where X is a large expression, with $X - X$ itself being buried inside a large expression. Here *HIST* has a cardinality of 1000, and evaluating *HIST*−*HIST* means reading through *HIST* once (it is only once, because of buffering). This takes 2 seconds of CPU time and overhead of 0.2 s for the mini-optimizer, 0.7 s for the midi-optimizer, and 0.8 s for the maxi-optimizer. These latter times are a pure overhead, because no access to the data base is made.

Query 2 involves three projections. The relation *HIST* has the three components (name, address, borrower's number) projected out, and from this (name, borrower's number) are chosen, followed by a final selection of (name). This kind of query has been found when using a position-independent query language with component names rather than numbers; projections are inserted in the translation into the ISBL language that we have used,

Table 6 Relations used in tests of optimizers.

Relations			*Components*			
name	*degree*	*cardinality*	*1*	*2*	*3*	*4*
ACQ	4	1000	acquisition number sequence 1 to 1000	author uniform 1, 10000	title uniform 1, 100000	Dewey code uniform 1, 200
BRW	4	10	borrower number sequence 1 to 10	name uniform 1, 10000	address uniform 1, 10000	status uniform 1, 10000
DDC	2	200	Dewey code sequence 1 to 2000	subject uniform 1, 10000		
HIST	4	1000	acquisition number uniform 1, 1000	borrower number uniform 1, 10	date in uniform 1, 10000	date out uniform 1, 10000

Table 7 Expressions used as queries in testing optimizers. The original expressions are shown together with the expressions to which they are transformed by the three levels of optimizer.

Test no.	*Level*	*Expression*
1	original	$HIST - HIST$
	Mini	$HIST - HIST$
	Midi	$\varnothing$
	Maxi	$\varnothing$
2	original	$((HIST\ \%\ C_2, C_3, C_1)\ \%\ C_1, C_3)\ \%\ C_2$
	Mini	$HIST\ \%\ C_1$
	Midi	$HIST\ \%\ C_1$
	Maxi	$HIST\ \%\ C_1$
3	original	$(ACQ \cup (BRW \cup HIST)) : C_3 < 3$
	Mini	$ACQ : C_3 < 3 \cup (BRW : C_3 < 3 \cup HIST : C_3 < 3)$
	Midi	$ACQ : C_3 < 3 \cup (BRW : C_3 < 3 \cup HIST : C_3 < 3)$
	Maxi	$ACQ : C_3 < 3 \cup (BRW : C_3 < 3 \cup HIST : C_3 < 3)$
4	original	$(ACQ \cup (BRW \cup HIST)) : C_3 > 2$
	Mini	$ACQ : C_3 > 2 \cup (BRW : C_3 > 2 \cup HIST\ ; C_3 > 2)$
	Midi	$ACQ : C_3 > 2 \cup (BRW : C_3 > 2 \cup HIST : C_3 > 2)$
	Maxi	$ACQ : C_3 > 2 \cup (BRW : C_3 > 2 \cup HIST : C_3 > 2)$
5	original	$ACQ * HIST : (C_1 = C_5\ \&\ C_2 < C_7\ \&\ C_3 < 100\ \&\ C_6 > 8\ \&\ (C_7 < 100 \mid C_3 < 100)$
	Mini	$ACQ * HIST : (C_1 = C_5\ \&\ C_2 < C_7\ \&\ C_3 < 100\ \&\ C_6 > 8\ \&\ (C_7 < 100 \mid C_3 < 100)$
	Midi	$((ACQ : C_3 < 100) * (HIST : 8 < C_2)) : C_1 = C_5\ \&\ C_2 < C_7\ \&\ (C_7 < 100 \mid C_3 < 100)$
	Maxi	$HIST : 8 < C_2 \rightarrow TEMP$ $((ACQ : C_3 < 100) * TEMP) : C_1 = C_5\ \&\ C_2 < C_7\ \&\ (C_7 < 100 \mid C_3 < 100)$
6	original	$((ACQ \cup BRW) \cap (BRW \cup HIST))\ \%\ C_1 - ((ACQ \cup BRW) \cap (BRW \cup HIST))\ \%\ C_2$
	Mini	$((ACQ \cup BRW) \cap (BRW\ HIST))\ \%\ C_1 - ((ACQ \cup BRW) \cap (BRW \cup HIST))\ \%\ C_2$
	Midi	$((ACQ \cup BRW) \cap (BRW \cup HIST))\ \%\ C_1 - ((ACQ \cup BRW) \cap (BRW \cup HIST))\ \%\ C_2$
	Maxi	$((ACQ \cup BRW) \cap (BRW \cup HIST)) \rightarrow TEMP1$ $TEMP1\ \%\ C_2 \rightarrow TEMP2$ $TEMP1\ \%\ C_1 \rightarrow TEMP3$ $TEMP2 - TEMP3$
7	original	$(((BRW * HIST * ACQ * DDC) : C_1 = C_6\ \&\ C_5 = C_9\ \&\ C_{12} = C_{13})\ \%\ C_1, C_2, C_3, C_4, C_{14}) : C_5 = 3927)$ $\%\ C_1, C_2, C_3, C_4$ $\cap\ (((BRW * HIST * ACQ * DDC) : C_1 = C_6\ \&\ C_5 = C_9\ \&\ C_{12} = C_{13})\ \%\ C_1, C_2, C_3, C_4, C_{14}) : C_5 = 9315)$ $\%\ C_1, C_2, C_3, C_4$
	Mini	$((BRW * HIST * ACQ * DDC) : (C_1 = C_6\ \&\ C_5 = C_9\ \&\ C_{12} = C_{13})\ \&\ C_{14} = 3927)\ \%\ C_1, C_2, C_3, C_4$ $\cap\ ((BRW * HIST * ACQ * DDC) : (C_1 = C_6\ \&\ C_5 = C_9\ \&\ C_{12} = C_{13})\ \&\ C_{14} = 93\ \%\ C_1, C_2, C_3, C_4$
	Midi	$(((BRW * HIST : C_1 = C_6) * ACQ : C_5 = C_9) * (DDC : C_2 = 3927) : C_{12} = C_{13}\ \%\ C_1, C_2, C_3, C_4$ $\cap\ (((BRW * HIST : C_1 = C_6) * ACQ : C_5 = C_9) * (DDC : C_2 = 9315) : C_{12} = \%\ C_1, C_2, C_3, C_4$
	Maxi	$DDC : C_2 = 9315 \rightarrow TEMP1$ $DDC : C_2 = 3927 \rightarrow TEMP2$ $(BRW * HIST : C_1 = C_6) * ACQ : C_5 = C_9 \rightarrow TEMP3$ $(TEMP3 * TEMP2 : C_{12} = C_{13})\ \%\ C_1, C_2, C_3, C_4 \cap (TEMP3 * TEMP1 : C_{12} = C_{13})\ \%\ C_1, C_2, C_3, C_4$

Table 8 Times taken to answer the queries of Table 7 without optimizers and with the three levels of optimizer. Times shown are the smallest of a series of trials (excepting query 7, in which only one trial was made). The measurements were made on an IBM System 370, model 145 using multi-access system CMS, with between 10 and 15 active users during the trials.

Query no.	*CPU time taken (s)*			
	no-opt	*Mini*	*Midi*	*Maxi*
1	2.01	2.21	0.73	0.81
2	10.37	1.20	1.69	2.22
3	3.78	2.73	2.74	2.95
4	5.49	5.73	5.83	8.41
5	4.13	4.36	3.60	3.76
6	3.87	4.12	4.25	3.23
7	estimated 10 days	estimated 10 days	1401	693

and three projections in succession are not unusual. On this particular query each projection causes a sort of 1000 degree-4 tuples, but when the projections are combined into a single projection, no sorting is necessary. This accounts for the considerable savings with all levels of optimizer. The result for the maxi-optimizer is much higher than that for the midi-optimizer because the maxi-optimizer does, in fact, store the result unnecessarily in this pilot implementation.

Thus query 2 is an example in which an end user or language translator can be allowed to generate code with no thought of performance. Query 7 is a second example.

Queries 3 and 4 test the movement of selections through set operations. Relations *ACQ*, *BRW*, and *HIST* are combined, and a subset of the result is selected as a function of the values in component 3. From the distributions shown in Table 6, it is seen that in query 3 the relations are drastically reduced in cardinality, but in query 4 very little reduction in cardinality occurs—hence the small improvement in 3 and small degradation in 4. Note that had the selection been on component 1 and not component 3, the times after transformation would have been significantly reduced due to the use of indexes on component 1 (the relations are stored using an indexed sequential organization). For example, the mini-optimizer time for $(ACQ \cup BRW \cup HIST)) : C_1 < 3$ is 1.94 s.

Query 5 demonstrates the movement of a selection through a join. Relation *ACQ* is joined to *HIST* on equal values in their respective first components, acquisition number (C_5 in the result is C_1 from *HIST*!), and various other conditions are imposed to select a subset of the result. The savings are small, because the join is an order n "equi-join," with equality on the leading components of relations *ACQ* and *HIST*. An n^2 join, for example, with $C_2 = C_6$, would have produced much more dramatic savings. Query 7 is such an example. For query 5, the maxi-optimizer uses an intermediate relation *TEMP* for the second argument to the join—this is simply making explicit what happens anyway with the midi-optimizer. The difference between the maxi- and midi-levels of optimizer is due, rather, to extra overhead, plus the redundant storage of the final result.

Query 6 is an example in which evaluating common subexpressions must always improve things. This is seen to be the case, although the savings are not significantly large. The example is highly artificial: a set combination of *ACQ*, *BRW*, and *HIST* is projected onto C_1 and onto C_2 to produce two intermediate relations, which are differenced.

Query 7 is adapted from a query that could arise in practice. Such a large expression would not be entered in one step—it would be built up in stages as a result of the deferred evaluation mechanism, as the user formed "views" of the data. In this example the requirement is to find out who are both mathematics readers and computing readers in the library. Initially all the information in the library data base was joined together and the interesting components (C_1, C_2, C_3, C_4) projected out. Then the readers of mathematics ($C_5 = 39A$) and computing ($C_5 = 9315$) were defined. The final intersection determines the people common to the two sets of readers. This lead to query 7, which was not practical to answer without some transformation. Moving the selection through the joins was sufficient to make the query feasible, with common subexpression evaluation and intermediate storage giving only an extra twofold improvement over the situation obtained with the midi-optimizer. This spectacular saving arises due to the removal of the extremely large joins implied by the query. This particular query arose because of the extremely flexible query formulation capabilities of PRTV—any system that permitted such queries would have to make similar transformations.

From these tests we conclude that the transformations do catch the extreme cases without degrading well formulated queries. In practice the sizes of relations would be orders of magnitude larger, and the savings would be more significant. The overhead for the optimizer would become completely insignificant. From Table 8 we see that the midi-optimizer level is the best, with possibly a conditional continuation to full common subexpression exploitation for large expressions.

These tests do, however, suggest that more attention is concentrated on the expensive operations, projection and join. The injection of sorts at a join to change a "quadratic join" of order n^2 into an equi-join of order n is worth exploring, as is the delaying of projections until

the last moment possible. These ideas will be taken up in later studies of data base optimization.

Conclusions

In this paper we have examined a particular data base problem, the amelioration or "optimization" of access to data in a relational data base. Only single accesses (single queries) have been considered. In this respect the problem studied is similar to that studied elsewhere, in DIAM [9], in language processing [10], and generally within relational systems [11–13].

In this paper we saw a series of transformations that can be applied to a relational expression to produce an equivalent expression or sequence of expressions that can be executed faster. These transformations were ameliorating rather than optimizing, because they could not be guaranteed to improve the time taken to compute the result. However, we argued that the chosen transformations were reasonably likely to improve performance.

We then saw the results of experiments on a prototype relational data base system and saw how well the transformations performed within the prototype when accessing a data base of random test data. These experiments showed the essential validity of the approach taken. At the cost of a small overhead that would become negligible for large data bases, severely bad expressions were recognized and transformed into acceptable expressions, whereas other expressions were in general improved. We saw one example in which the performance was degraded by the transformations.

It is clear that further study is required, because there are transformations involving projections that may well be worth making. It is also worth considering the longer term storage of intermediate results, using these in later expressions if possible. The methods used for common subexpression detection provide the techniques necessary for determining if suitable intermediate results have been previously computed.

Acknowledgments

The author thanks T. Rogers and S. Todd for numerous discussions that have helped to clarify problems and methods.

Appendix: Cost of evaluating an expression

What does it cost to evaluate a given relational expression E? Evaluation means successively producing all the complete tuples of the relation at some location within main storage. If the expression E is of the form F op G, then clearly the cost of evaluating E would be the cost of evaluating F and G plus some extra cost for evaluating the final operation. Thus we expect

$$c(E = F \text{ op } F) = c(F) + c(\text{op}, F, G).$$

This decomposition of the cost of evaluating the expression E is true for the PRTV implementation but could conceivably not be true for some unusual implementation of a relational data base system.

For relations that are explicitly stored on disk, their cost is simply the cost of moving them into main storage from disk. Thus for stored relation A we would have

$$c(A) = \delta \,.n(A).d(A),$$

where δ is some constant associated with I/O, $n(A)$ is the cardinality of A, and $d(A)$ is the degree of A. The constant δ is the cost of reading in one component of one tuple of the relation. In the PRTV system this cost calculation is complicated by compression techniques used for disk storage, but here we overlook this consideration.

Operations themselves are CPU bound. We take this into account by assuming tuple at a time evaluation without the explicit storage of intermediate results. Let us look at PRTV set union. Suppose that we have two relations A and B, either stored on disk or produced by the evaluation of two expressions. We suppose that these relations are produced a tuple at a time for input to the union procedure. They are produced in the same sort order, so that the union operation can be performed by a simple merge of the two sorted sequences. The union procedure also produces a sequence of tuples, one at a time, in the same sort order. The code for UNION might look as follows:

```
FIRST: X = FIRST TUPLE OF (A);
       Y = FIRST TUPLE OF (B);
NEXT:  CASE
       X = Y     OUTPUT(X); X = NEXT
                            TUPLE OF (A);
                            Y = NEXT
                            TUPLE OF (B);
       X < Y     OUTPUT(X); X = NEXT
                            TUPLE OF (A);
       X > Y     OUTPUT(Y); Y = NEXT
                            TUPLE OF (B);
       ESAC;
```

We assume that each relation is terminated with some "infinitely large" value, so that the merge continues to completion. Let us estimate the cost of the union from this code. The three alternatives within the CASE statement are executed respectively $n(A \cap B)$, $n(A) - n(A \cap B)$, $n(B) - n(A \cap B)$ times. Suppose that the first path costs σ units per component per tuple, and the second and third cost ρ units. Then the total cost is

$$c(\text{UNION}, A, B) = n(A \cap B)\sigma d + [n(A) - n(A \cap B)]\rho d + [n(B) - n(A \cap B)]\rho d,$$

where d is the degree of the relation. Using the identity

$$n(A \cup B) = n(A) + n(B) - n(A \cap B),$$

and substituting

$$\alpha = \sigma - \rho, \beta = 2\rho - \sigma,$$

we find that

$$c(\text{UNION}, A, B) = \alpha[n(A) + n(B)]d + \beta n(A \cup B)\ d.$$

We assume that the costs of all operations take the general form

$$c(\text{op}, A, B) = \alpha_{\text{op}}[n(A)\ d(A) + n(B)\ d(B)] + \beta_{\text{op}} n(A\ \text{op}\ B)\ d(A\ \text{op}\ B)$$

for suitable constants α_{op} and β_{op}. The important point to note about these cost functions is that they are functions of cardinality, and in general the cardinality is not known.

Obviously if the operands A and B are explicitly stored, their cardinality is known, but what is the cardinality of $A \cup B$, $A \cap B$, etc.? All we know is

$$\max[n(A), n(B)] \leq n(A \cup B) \leq n(A) + n(B),$$
$$0 \leq n(A \cap B) \leq \min[n(A), n(B)].$$

The variability in the cardinality of the result can be considerable. And when the operands A and B are themselves the result of evaluating expressions, the variability is much worse.

The cost of evaluating a complete expression includes an I/O component associated with the input of the stored relations, the storage of the result (if this is done), and the storage of the intermediate results that require storage. There is a CPU component, partly associated with each operation and partly with the access to disk. All components of the cost depend upon the cardinalities of the relations being processed by that part of the expression. Only the cardinalities of the stored relations are known precisely, and to estimate the cost of evaluating an expression we must be able to estimate the cardinalities of the various relations formed during the evaluation.

How do we estimate cardinality? As we have seen, the cardinality of the result of as simple an operation as intersection can vary from zero to the smaller of the two cardinalities of the operands. What precisely happens depends upon the detailed content of the operand relations. To estimate the cardinality we could do something very crude, such as

$$n(A \cap B) = \theta \min[n(A), n(B)],$$

where θ is a "suitable" constant. This may be easy to calculate, but it is not very satisfactory. To do the estimation adequately, we really need to store information about the distribution of data within the relations as samples from some underlying common population.

In the PRTV optimizer, cost estimates have been avoided where possible. However, sometimes cost estimates are necessary, in which case they are computed recursively using the above equations, calculating cardinalities by this very crude method. Generally, our optimization methods are not based on cost estimates, but rather on transformations that can be guaranteed to improve performance or that can be expected usually to improve performance on the basis of some heuristic argument. Only in special cases are cardinality and cost estimates used to guide decisions.

Although the estimation of the cost of evaluating an expression has been abandoned for this paper, it remains an important problem, not only for optimization. It is also desirable to be able to estimate costs of queries so that they can be referred back to a user before actually undertaking the work (see, for example, [14]). Any cost estimate to within an order of magnitude would be better than none.

References

1. S. J. P. Todd, "PRTV, A Technical Overview," *Report UKSCOO75*, IBM UKSC, Peterlee, Co. Durham, England, 1975. (A summary was presented at the ACM conference on Very Large Data Bases, Boston, September 1975.)
2. L. Aspinall, "Data base reorganization—algorithms," *Report UKSCOO29*, IBM UKSC, Peterlee, Co. Durham, England, February 1972.
3. L. Aspinall, C. J. Bell, and T. W. Rogers, "Data base reorganization concepts," *Report UKSCOO11*, IBM UKSC Peterlee, Co. Durham, England, February 1972.
4. C. J. Bell, B. K. Aldred, and T. W. Rogers, "Adaptability to change in large data base information retrieval systems," *Report UKSC0027*, IBM UKSC, Peterlee, Co. Durham, England, April 1972.
5. R. G. Casey and I. Osman, "Generalised Page Replacement Algorithms in a Relational Data Base," *Report UKSC0056*, IBM UKSC, Peterlee, Co. Durham, England, April 1974.
6. P. A. V. Hall, "Common Subexpression Identification in General Algebraic Systems," *Report UKSC0060*, IBM UKSC, Peterlee, Co. Durham, England, November 1974.
7. P. A. V. Hall and S. J. P. Todd, "Factorisations of Algebraic Expressions," *Report UKSC0055*, IBM UKSC, Peterlee, Co. Durham, England, April 1974.
8. R. Sethi, "Testing for the Church-Rosser Property," *J. ACM*, **21**, 671 (October 1974).
9. S. P. Ghosh and M. E. Senko, "String Path Search Procedures for Data Base Systems," *IBM J. Res. Develop.* **18**, 408 (1974).
10. B. M. Leavenworth and J. E. Sammet, "An Overview of Nonprocedural Languages," *SIGPLAN Notices* (ACM) **9**, 1 (April 1974).
11. I. M. Osman, "Matching Storage Organization to Usage Pattern in a Relational Data Base," Ph.D. Thesis, University of Durham, Durham, England, 1974.
12. F. P. Palermo, "A Data Base Search Problem," *Proceedings Fourth Int. Symp. on Computer and Information Sci-*

ence, Miami Beach, 1972. (Also *Research Report R J 1972*, IBM Research Laboratory, San Jose, California 95193.)
13. J. M. Smith and P. Y. T. Chang, "Optimizing the Performance of a Relational Algebra Data Base Interface," *Comm. ACM* **18**, 568 (1975).
14. P. M. Stocker and P. A. Dearnley, "Self-organizing data management systems," *Computer Journal* **16**, 100 (1973).

Received June 6, 1975; revised September 22, 1975

The author, who was assigned to the United Kingdom Scientific Center, Peterlee, England, when this work was done, is now at the British Ship Research Association, Wallsend, Tyne and Wear, England.