EFFICIENT EVALUATION OF EXPRESSIONS IN A RELATIONAL ALGEBRA

Robert M. Pecherer
Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California, Berkeley, California 94720
(415) 454-5427

The retrieval of information from a relational data base is treated as the evaluation of relation-defining expressions in an algebra over the data base relations. An algebra is introduced that offers computational advantages over previously defined relational algebras. A very general storage model is assumed which admits a wide variety of storage structures. An algorithm for $O(n\log_2 n)$ division is described, and a rapid evaluation schema for a large and important class of expressions is derived. Finally, techniques are explored for translating an algebraic expression into an equivalent expression that is inherently "quicker" to evaluate.

I. INTRODUCTION

In a relational data base system, the user logically views large quantities of formatted data as stored in time-varying, finite relations of assorted degrees. Updates to and retrievals from his collection of data are often specified in a descriptive, non-procedural language such as DSL-ALPHA [1], SEQUEL [2], SQUARE [3], DAMAS [4], or QUEL [5]. The physical representation of user relations and procedural solutions to non-procedural user requests are the main problem of the system designer, and as yet, efficient implementation techniques are largely unexplored. The presence of multiple users, multiple views and security and integrity constraints complicate the problem and are not discussed. Here, we consider only the formation of response relations (retrievals), so that the time-varying nature of the data base is ignored.

At present, there are three main approaches to the decomposition of non-procedural retrievals:

(1) Iterative decomposition by tuple subsitution [5]. In the retrieval statement, the tuples of one relation are substituted to produce one or more simpler statements with data values replacing variables. By repeated application, all variables are removed and the statements can be directly processed. This approach is further discussed in [4, 5, 6].

(2) Translation to an algebraic expression employing relational operators on the data base relations. By implementing the operators, the expression is evaluated to produce the response relation. This suggestion is pursued in [7, 8].

(3) Compilation to a set of procedures in a hierarchical or network-oriented language. This approach is followed in DIAM [9].

The first method has been implemented in the INGRES [5, 6] project at Berkeley. The third approach has been proposed by Senko [9] and by Tsichritzis [10]. The feasibility of the second approach has been demonstrated with a translator from the non-procedural language ALPHA to a relational algebra by Codd [7], and an APL implementation of this algebra by Palermo [8]. This paper investigates efficiency considerations for the second approach, building on Palermo [8]. Results are applicable to algebraic expressions generated by the Codd Reduction Algorithm (CRA) [7] and to algebraic expressions in general. The reader is assumed to be familiar with [7], especially the CRA and the retrieval language ALPHA.

We are concerned here with 2 factors:

(i) efficient implementation procedures for the operators of a relational algebra, and

(ii) translation of algebraic expressions to equivalent algebraic expressions which are inherently "quicker" to evaluate.

The paper is divided into 4 sections. The first introduces definitions and terminology, and describes the relational algebra whose implementation we study here. The second discusses techniques for the efficient implementation of this algebra; results are presented for arbitrary algebraic expressions and for the class of expressions produced by the CRA. The third describes a number of efficiency results obtainable by translation to equivalent expressions, and the last section offers a summary and conclusion.

II. DEFINITIONS AND TERMINOLOGY FOR A RELATIONAL ALGEGRA

The definitions and terminology are similar to Codd [1, 7, 11, 12]. The relational algebra defined here is derived from [7], but employs fewer, more powerful operators; the differences will be noted.

Relations

A domain D is set; a domain is simple if its members are not themselves sets. Let $D_1,\ldots,D_n$ be domains. A relation R over $D_1,\ldots,D_n$ is a subset of the Cartesian Product $D_1\times\ldots\times D_n$ and is said to be an n-nary relation or relation of degree n. The members of R are called n-tuples or simply tuples. If r is an n-tuple, r[i] designates the i^{th} domain value or i^{th} attribute of r, for $1 \le i \le n$. If $A = a_1,..,a_k$ is a list of integers such that $1 \le a_j \le n$ for $j = 1,\ldots,k$, then A is a domain identifying list for any n-nary relation, and r[A] designates the k-tuple $(r[a_1],\ldots,r[a_k])$. If r is an n-tuple and s an m-tuple, $r\frown s$ designates the n+m-tuple (r[1],...r[n],s[1], ...,s[m]). A relation is first-normal or normal if each of its domains is simple. All relations considered here are assumed to be normal. In the sequel, deg(R) denotes the degree of relation R; the size of a relation is the number of tuples it contains.

A Relational Algebra

Any relational algebra consists of operators which map relations or pairs of relations to relations, and a set of relations closed under the operators. The following definitions apply only to normal relations. Note that tuple attributes are identified by position.

Definition (2.1) Let L be a domain identifying list for relation R. The projection $\pi_L(R)$ is defined by:
$\pi_L(R) = \{r[L]: \ r \in R\}$.

Definition (2.2) Let R and S be relations of degree n and m respectively. The product R*S is a set of n+m-tuples defined by:
$R*S = \{\widehat{r\,s}: \ r \in R \wedge s \in S\}$.

Definition (2.3) Let R and S be relations, and r and s tuple variables for R and S respectively. Let f be a 0-1 function specified as a Boolean combination of terms of the form

$$x \theta y$$

where θ is one of $\{=, \neq, <, \leq, >, \geq\}$, and each x and y is an arithmetic expression involving tuple attributes r[i] and s[j], simple functions (such as log, sin, etc.) of attributes, and constants. The join R[f]S is the set of tuples in R*S for which the function f is equal to "1." e.g.,
$R[f]S = \{\widehat{r\,s}: \ r \in R \wedge s \in S \wedge f(r,s) = 1\}$.

In (2.3), the symbols "r" and "s" always refer to the left and right operands respectively in the binary join; this is simply a syntactic convenience.

Definition (2.4) Let R be a relation, r a tuple variable for R, and g a Boolean function such as "f" in join, but referencing only attributes of r. The restriction R[g] is the set of tuples in R for which g is equal to "1."
$R[g] = \{r: \ r \in R \wedge g(r) = 1\}$.

Definition (2.5) Let R and S be relations of degree n and m respectively; let A and B be domain identifying lists for R and S respectively, both of length k<n and both without repetition. Let $\overline{A}$ be a domain identifying list for R complementary to A and in increasing order (viz. n=5, A=1,4,3 then $\overline{A}$=2,5). The division R[A÷B]S designates a subset of n-k-tuples in $\pi_{\overline{A}}(R)$ as follows:

$$R[A \div B]S = \{r[\overline{A}]: \ r \in R \wedge \forall s \in S \ \exists r' \in R \ [r[\overline{A}] = r'[\overline{A}] \wedge r'[A] = s[B]]\}$$

A more revealing definition and numerous examples of division are given in [7]; a minor difference is that in our definition, when S is vacuous, $R[A \div B]S = \pi_{\overline{A}}(R)$, whereas in Codd's, R[A÷B]S is the empty relation. A third possibility is to make division by a vacuous relation undefined. Our definition is more consistent with the intended correspondence between division in the algebra and testing for universal quantification in a first-order predicate calculus (see [7, 8] for a discussion). For our purposes, algebraic expressions are assumed to be over non-empty relations, so that the difference in definition is of no concern.

Comparing the above definitions to Codd's Relational Algebra [7], projection and product are identical, as is division when the divisor relation is non-empty. For join and restriction, Codd's notation is such that the only Boolean functions allowed are conjunctions of θ-comparisons of attribute values, with all θ's the same. Our definitions are motivated by the belief that the relations of an implemented system will reside in a slow secondary memory, and the input of tuples from R and S to form R[f]S (or from just R to form R[g]) is expected to be more time-consuming than application of the function f (or g). So that the retrieval of (say)

$$\{\widehat{r\,s}: \ r \in R \wedge s \in S \wedge (r[1]=s[1] \vee r[2] \neq s[2])\}$$

may be expected to proceed more quickly by evaluating

$$R[r[1] = s[1] \vee r[2] \neq s[2]]S$$

than by evaluating

$$R[r[1] = s[1]]S \quad \text{and} \quad R[r[2] \neq s[2]]S$$

and forming their union. A similar argument applies for restriction.

Codd's algebra includes the set operations of union, intersection and relative complement; the following identities indicate that the latter 2 are not necessary:

$$(2.6) \quad R \cap S = \pi_{1,\ldots,n}(R[r[1] = s[1] \wedge \ldots \wedge r[n] = s[n]]S)$$

$$(2.7) \quad R \setminus S = (R[r[1] \neq s[1] \vee \ldots \vee r[n] \neq s[n]]S)[n+1,\ldots,2n \div 1,\ldots,n]S$$

where deg(R) = deg(S) = n and S is non-empty

Union cannot be obtained from the other operators since it produces relations with domains which are supersets of the operand relation domains, and none of the other operators can achieve this. Allowing disjunctions in the definitions of the Boolean functions "f" and "g" (for join and restriction) eliminates the primary need for union. If the union of two compatible relations must be formed, we presume this to take place external to the algebra; implementation of union will not be discussed.

The five operators defined here are not a minimal set since

$$(2.8) \quad R*S = R[r[1] = s[1] \vee r[1] \neq s[1]]S, \text{ and}$$

$$(2.9) \quad R[g] = \pi_{1,\ldots,n}(R[r[1] = s[1] \wedge \ldots \wedge r[n] = s[n] \wedge g]R)$$

where deg(R) = n.

No computational advantage is indicated by eliminating product and restriction operators, and in fact, implementation of restriction of a relation by a projection of a join of the relation with itself is probably a poor idea.

In the next section, a storage framework for relations and time approximations for evaluation of the operators are introduced. A fast way to perform division is derived, and a very fast evaluation technique for an important class of algebraic expressions is demonstrated.

III. EFFICIENCY IN THE IMPLEMENTATION OF RELATIONAL OPERATORS

In this section, we proceed as if our only goal is the evaluation of syntactically correct expressions in the relational algebra over a fixed set of stored data base relations $R_1,\ldots,R_p$ of size $n_1,\ldots,n_p$. Correct evaluation of each operator requires examination of every tuple of each operand relation, however, since a relation is a set with no specified order properties, the order in which tuples are examined is of no logical consequence to the result. For the evaluation of certain operators, the retrieval order of tuples in the operand relation(s) can affect the time required. To demonstrate this, we make the following assumptions:

(3.1) All relations (data base, intermediate, result) are maintained in a slow secondary memory as tables of tuples; the order in which they are stored is the only order in which they can be retrieved.

(3.2) All tuples of a relation are encoded in a fixed-length field. The time required to retrieve every tuple of a single relation is proportional to its size.

(3.3) Application of Boolean functions to tuples is performed in primary memory by a single processor in an amount of time which is insignificant compared to the input time for the operand relations or the output time for the result.

We are concerned with the relative efficiency of evaluating the operators for different retrieval orders. We approximate the amount of time required as a function of the size of the operands. When the time is specified as "$0(f(n))$," this indicates that the actual time $T(n)$ is proportional to $f(n)$, or equivalently,

$$(3.4)\quad \lim_{n\to\infty} T(n)/f(n) = K$$

where K is a proportionality constant (nonzero).

The evaluation of $R_i[g]$ requires time $0(n_i)$ since every tuple of R_i must be retrieved for testing with g. The evaluation of R_i*R_j requires time $0(n_i*n_j)$ since n_i*n_j tuples must be output. We assume that evaluation of $R_i[f]R_j$ also requires time $0(n_i*n_j)$ since every tuple of R_i*R_j must be formed and tested by f. These values are independent of the retrieval order of the operand relations except for certain simple Boolean functions which do not concern us here. We are concerned with projection and division, for which different (known) storage orders require different evaluation times.

The evaluation of $\pi_L(R_i)$ (when L is not a permutation of the sequence "$1,\ldots\deg(R_i)$") is complicated by the fact that the result is a set without duplicate tuples. If it is known that R_i is sorted on the domains of L, all tuples r in R_i with the same value of r[L] appear sequentially when R_i is retrieved. The storage of duplicate tuples can be avoided with comparisons between consecutively retrieved tuples (not by scanning all previously stored tuples), so in this case, $\pi_L(R_i)$ can be evaluated in time $0(n_i)$. The best method we have found for evaluating $\pi_L(R_i)$ when R_i is unsorted is by a simultaneous sort-and-project procedure that eliminates domains not in L and duplicate tuple values. This method is dominated by the sorting time which we take to be $0(n_i\log_2 n_i)$ (see Knuth [13], pp. 361-376). Under the given assumptions, we have that:

(3.5) $\pi_L(R_i)$ can be evaluated in time $0(n_i)$ if each subset of R_i with the same projection r[L] is consecutively retrievable.

The amount of time to evaluate $R_i[A\div B]R_j$ depends not only on n_i and n_j, but also on the size of the result. In the following derivation, we assume $n_i \geq n_j$, and that s[B] assumes n_j distinct values as s ranges over R_j.

From the definition, we have that for each r in R_i:

$$(3.6)\quad r[\overline{A}] \in R_i[A\div B]R_j \Leftrightarrow \forall s \in R_j\ \exists\ r' \in R_i[r[\overline{A}] = r'[\overline{A}] \wedge r'[A] = s[B]]$$

$$(3.7)\quad r[\overline{A}] \notin R_i[A\div B]R_j \Leftrightarrow \exists\ s \in R_j \forall r' \in R_i[r[\overline{A}] \neq r'[\overline{A}] \vee r'[A] \neq s[B]]$$

To verify the condition on the right hand side (rhs) of (3.6) requires time $n_j*0(n_i)$ since there are n_j tuples in R_j, and it requires time $0(n_i)$ to find r' in R_i. To verify the condition on the rhs of (3.7) requires time $0(n_i)$ since every tuple of R_i must be checked. If m is the size of the result, m is bounded from below by 0 and above by n_i/n_j since for each t in the result, there must be n_j tuples $z_1,\ldots,z_{n_j}$ in R_i with $z_1[\overline{A}] = \ldots = z_{n_j}[\overline{A}] = t$, and these tuples cannot satisfy the necessary condition for any other t' in $R_i[A\div B]R_j$. This yields the time approximation:

$$(3.9)\quad m*n_j*0(n_i) + (n_i - m)*0(n_i)$$

for the total evaluation, which is $0(n_i^2)$ over the range of m. Clever programming can produce smaller values for K in (3.4) for division, but the $0(n_i^2)$ approximation is valid.

When it is known that R_i is sorted on $\overline{A}$, A and R_j on B, then for each t in $R_i[A\div B]R_j$, (at least) n_j consecutively retrievable tuples in R_i will have $z[\overline{A}] = t$, and within this subset, the n_j tuples that satisfy the role of r' in the rhs of (3.6) will appear in the same order as the n_j tuples of R_j. The following example illustrates this:

(3.9) Example: $R_i[2,3\div 1,2]R_j$

(unsorted R_i, R_j)

R_i	1	2	3	R_j	1	2
	A	X	1		Z	3
	C	Z	3		X	1
	B	Z	3			
	C	X	1			
	A	Z	1			
	A	Z	3			
	B	X	2			
	B	Z	1			

(R_i sorted on 1,2,3; R_j sorted on 1,2)

R_i	1	2	3	R_j	1	2
	A	X	1		X	1
	A	Z	1		Z	3
	A	Z	3			
	B	X	2			
	B	Z	1			
	B	Z	3			
	C	X	1			
	C	Z	3			

Having achieved this arrangement of the operand relations, evaluation of $R_i[A\div B]R_j$ can be performed with a sequential scan of R_i since no tuple of R_i has to be compared to more than one tuple of R_j. The cost of sorting both R_i and R_j is $0(n_i\log_2 n_i)$

since $n_i \geq n_j$, and since the division can then be performed in $O(n_i)$, the total time is $O(n_i \log_2 n_i)$, which is superior to $O(n_i^2)$ for unordered R_i and R_j. For an arbitrary algebraic expression, this technique and the sort-project technique for projection guarantee that any projection or division can be evaluated in time $O(n\log_2 n)$ where n is the size of the projected or divided relation. And under the given assumptions, we also have that:

(3.10) $R_i[A \div B]R_j$ can be evaluated in $O(n_i)$ if each subset of R_i with the same projection $r[\overline{A}]$ is consecutively retrievable, and the tuples of each subset are retrievable in the same order as the tuples of R_j.

For an important class of algebraic expressions involving projections and divisions, the conditions in (3.5) and (3.10) can be achieved in the intermediate results of evaluation at no extra cost. The Codd Reduction Algorithm (CRA) generates the class from a restricted set of ALPHA expressions (see [7] for details of the CRA and ALPHA); the general form is:

$$(3.11)\quad \pi_L(G_{k+1}(\ldots G_{k+q}((R_1*\ldots*R_{k+q})[g])\ldots))$$

in which $G_{k+j}(X)$ is either the projection

$$\pi_{1,\ldots m_{k+j}}(X)$$

or the division

$$X[m_{k+j}+1,\ldots,m_{k+j+1} \div 1,\ldots,\deg(R_{k+j})]R_{k+j}$$

where $m_i = \deg(R_1) + \ldots + \deg(R_{i-1})$.
We refer to such expressions as "CRA-expressions." Suppose the evaluation is performed by producing $T_0,\ldots,T_q,W$ where:

$$T_0 = (R_1*\ldots*R_k*\ldots*R_{k+q})[g]$$

$$T_1 = G_{k+q}(T_0)$$

$$\vdots$$

$$T_q = G_{k+1}(T_{q-1})$$

$$W = \pi_L(T_q)\ .$$

When g is identically true, $T_i = R_1*\ldots*R_{k+q-i}$ for $i = 0,\ldots,q$; each G_{k+j} produces $T_{q-j+1} = R_1*\ldots*R_{k+j-1}$ from $T_{q-j} = R_1*\ldots*R_{k+j}$. If T_0 is generated as $(R_1*\ldots*R_k)*(R_{k+1}*(R_{k+2}*(\ldots(R_{k+q}))))$, then for every $t \in T_1 = R_1*\ldots*R_{k+q-1}$, the subset $t*R_{k+q}$ appears sequentially in T_0, so the conditions in (3.5) are met; if R_{k+q} is unaltered, the conditions in (3.10) are also met. G_{k+q} can then be applied to T_0 in time proportional to its size. If R_{k+q-1} is unaltered and the tuples of T_1 are stored in the order that the tuples of T_0 are sequentially scanned, T_1 meets the conditions also, so again, G_{k+q-1} can be applied in time proportional to the size of T_1. Repeating the procedure for $G_{k+q-1},\ldots,G_{k+1}$, each such projection or division can proceed in time proportional to the size of the operand. When g is any Boolean function, the same argument applies to subsets of $R_1*\ldots*R_{k+q},\ldots,R_1*\ldots*R_{k+1}$, so that again, the projections and divisions proceed quickly. Only for the last projection "$W = \pi_L(T_q)$" can the condition in (3.5) fail.

The importance of this technique is that O(n) procedures for every division and projection (except the last) are available without any rearrangement of the data base relations. The CRA-expressions can represent complex queries over many relations; rapid evaluation is critical to the performance of the entire system.

IV. EFFICIENCY BY TRANSLATION TO AN EQUIVALENT EXPRESSION

Complex queries to an information retrieval system are so time-consuming that even if they represent a small fraction of the total queries, their effect could seriously degrade the response time for simpler requests. Examination of the general CRA-expression

$$(4.1)\quad \pi_L(G_{k+1}(\ldots G_{k+q}((R_1*\ldots*R_{k+q})[g])\ldots))$$

indicates 2 reasons why complex queries require so much time:

(i) The product relation $R_1*\ldots*R_{k+q}$ is exceedingly large, perhaps too large to be stored in secondary memory.

(ii) The restriction "g" and each projection and division require at least a sequential scan of every tuple left by the application of the previously applied operators.

The previous section discussed operator implementation; in this section, properties of the operators are exploited to produce equivalent expressions which reduce the size of the product space and eliminate certain projections and divisions. The translation of an expression to a more easily evaluated expression is expected to proceed mechanically, but guided by accumulated statistics on the data base relations and previous evaluations, and possibly with user interaction. This work was stimulated by Palermo [8].

Palermo [8] has shown that in (4.1),

(4.2) For $1 \leq i \leq k$, any domain of R_i not referenced in "L" or "g" can be projected out prior to evaluation without affecting the result.

(4.3) For $k < i \leq k + q$, any domain of R_i not referenced in "g" can similarly be projected out provided G_i is a projection and not a division.

This allows us to rewrite (4.1) as

$$(4.4)\quad \pi_{L'}(G'_{k+1}(\ldots G'_{k+q}((\pi_{L_1}(R_1)*\ldots*\pi_{L_{k+q}}(R_{k+q}))[g'])\ldots))$$

where each projection $\pi_{L_1}(R_1)$ eliminates domains in R_i in accordance with (4.2) and (4.3), and L', G'_{k+j} and g' are derived from L, G_{k+j} and g in (4.1) to reflect the altered positions of the relevant domains. The new expression (4.4) requires less space to evaluate, but the time to perform projections indicates that this transformation should be used cautiously.

[2] Chamberlin, D. D. and R. F. Boyce, "SEQUEL: A Structural English Query Language," IBM Research Laboratory, San Jose, CA

[3] Boyce, R. F., D. D. Chamberlin, M. M. Hammer and W. F. King, "Specifying Queries as Relational Expressions: SQUARE," IBM Technical Report RJ 1291, October 1973.

[4] Rothnie, J. B., "The Design of Generalized Data Management Systems," Ph.D. Dissertation, Dept. of Civil Engineering, MIT, Septe ber 1972.

[5] McDonald, N., M. Stonebraker and E. Wong, "Preliminary Design of INGRES," ERL, Univ. Calif., Berkeley, Memo #ERL-M435, April 1974.

[6] Stonebraker, M. and E. Wong, "INGRES-A Relational Data Base System," ERL, Univ. of Calif., Berkeley, Memo #ERL-M472, November 1974.

[7] Codd, E. F., "Relational Completeness of Data Base Sublanguages," Courant Computer Science Symposium 6, May 1972.

[8] Palermo, F. P., "A Data Base Search Problem," IBM Technical Report RJ 1072, July 1972.

[9] Senko, M. E., E. B. Altman, M. M. Astrahan and P. L. Fehder, "Data Structures and Accessing in Data-Base Systems," IBM Systems Journal, Vol. 12, No. 1, 1973.

[10] Tsichritzis, D., "A Network Framework for Relation Implementation," Proc. of the IFIPS TC-2 Working Conference on Data Definition Language, January 1975.

[11] Codd, E. F., "A Relational Model of Data for Large Shared Data Banks," CACM, Vol. 13, No. 6, June 1970.

[12] Codd, E. F., "Seven Steps to Rendezvous with the Casual User," IBM Research, San Jose, CA, RJ 1333, January 1974.

[13] Knuth, D. E., The Art of Computer Programming, Volume 3, Sorting and Searching, Addison-Wesley, 1973.