SIAM J. COMPUT.
Vol. 10, No. 3, August 1981

0097-5397/81/1003-0001 $01.00/0

INFERRING A TREE FROM LOWEST COMMON ANCESTORS WITH AN APPLICATION TO THE OPTIMIZATION OF RELATIONAL EXPRESSIONS*

A. V. AHO†, Y. SAGIV‡, T. G. SZYMANSKI† AND J. D. ULLMAN§

Abstract. We present an algorithm for constructing a tree to satisfy a set of lineage constraints on common ancestors. We then apply this algorithm to synthesize a relational algebra expression from a simple tableau, a problem arising in the theory of relational databases.

Key words. tree algorithms, lowest common ancestors, relational databases, relational algebra, tableaux, query optimization, join minimization

1. A tree discovery problem. In a rooted tree, the lowest common ancestor of two nodes x and y, denoted (x, y), is the node a that is an ancestor of both x and y such that no proper descendant of a is also an ancestor of both x and y. Suppose we are told a tree T has leaves numbered $1, 2, \cdots, n$, and we are given a set of constraints on the lowest common ancestors of certain pairs of leaves. In particular, suppose the constraints are of the form $(i, j) < (k, l)$ where $i \neq j$ and $k \neq l$, meaning that the lowest common ancestor of i and j is a proper descendant of the lowest common ancestor of k and l. Note that the order of i and j in (i, j) and of k and l in (k, l) are irrelevant. From a set of constraints of this form, can we reconstruct T, or determine that no such tree exists?

Example 1. Suppose we are given the constraints

$$\begin{aligned} &(1, 2) < (1, 3), \\ &(3, 4) < (1, 5), \\ &(3, 5) < (2, 4). \end{aligned} \tag{1.1}$$

One possible tree T consonant with these constraints is shown in Fig. 1. However, if we add the constraint $(4, 5) < (1, 2)$, then it can be shown that no tree can simultaneously satisfy all these constraints.

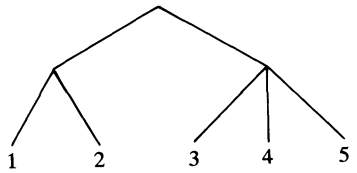

FIG. 1. *A solution to constraints* (1.1).

In this paper we give an efficient algorithm for solving this problem. We then present an application of this algorithm to the problem of synthesizing a relational algebra expression from a simple tableau.

2. Finding the tree. The central idea behind the solution is to determine for a potential tree T the sets of leaves that are descendants of each child of the root of T. Call these sets $S_1, S_2, \cdots, S_r$. We may assume $r \geqq 2$, since if a tree satisfying a set of

* Received by the editors July 10, 1979, and in revised form June 3, 1980.

† Bell Laboratories, Murray Hill, New Jersey 07974.

‡ Department of Computer Science, University of Illinois, at Urbana-Champaign, Urbana, Illinois 61801.

§ Department of Computer Science, Stanford University, Stanford, California. The work of this author was supported in part by the National Science Foundation under grant MCS-76-15255.

constraints exists, we can find another tree satisfying the same set of constraints if we merge each node having one child with that child. In Example 1, we have $S_1 = \{1, 2\}$ and $S_2 = \{3, 4, 5\}$.

There are two conditions that these sets must satisfy for each constraint $(i, j) < (k, l)$.

(1) i and j must be in the same set. Otherwise (i, j) is the root of T, and the root cannot be a proper descendant of (k, l).

(2) Either k and l are in different sets, or i, j, k and l are all together in one set. Otherwise (i, j) cannot be a proper descendant of (k, l).

We shall also show that conditions (1) and (2) are sufficient. Thus, if we can partition the nodes into two or more sets satisfying (1) and (2), and if we can recursively build trees for each set, then a tree exists; otherwise, one does not.

Given a set of constraints C, we define a partition π_C on the leaves $1, 2, \cdots, n$ using the following rules:

(1) If $(i, j) < (k, l)$ is a constraint, then i and j are in one block of π_C.

(2) If $(i, j) < (k, l)$ is a constraint, and k and l are in one block, then i, j, k and l are all in the same block.

(3) No two leaves are in the same block of π_C unless it follows from (1) and (2).

Example 2. π_C for the constraints given in Example 1 is $\{1, 2\}, \{3, 4, 5\}$. Note that no instance of rule (2) is used. If we add to (1.1) the constraint $(4, 5) < (1, 2)$, to obtain the set

$$(1, 2) < (1, 3),$$
$$(3, 4) < (1, 5),$$
$$(3, 5) < (2, 4),$$
$$(4, 5) < (1, 2),$$

then by rule (2) the two blocks must be merged, yielding a trivial partition. Since a necessary condition for the existence of a tree is that π_C have more than one block, we can immediately infer that this new set of constraints is not satisfied by any tree.

In Fig. 2 we present a recursive algorithm to build a tree T satisfying a set of constraints C on a nonempty set of nodes S. It returns the null tree if no tree exists. The basic idea is to compute the partition π_C, check that it has at least two blocks $S_1, S_2, \cdots, S_r$, $r \geqq 2$, and construct the sets of constraints C_m, $1 \leqq m \leqq r$, such that C_m is C restricted to those constraints that involve members of S_m only.

THEOREM 1. *If* BUILD(S, C) *returns a nonnull tree* T, *then* T *satisfies the constraint set* C.

Proof. We proceed by induction on the size of S. The basis, one node, is trivial. Suppose then that the theorem is true for all sets smaller than S. Because every recursive invocation of BUILD is applied to a set of strictly smaller size than S, we can assume that all recursive calls of BUILD obey the inductive hypothesis.

We must show that all constraints in C are satisfied by T. Accordingly, let $(i, j) < (k, l)$ be an arbitrary member of C. Two cases arise depending on whether k and l are in the same or in different blocks.

Case 1. If k and l are in the same block, then all of i, j, k and l are in some set S_m by rule (2) in the definition of π_C. Also, $(i, j) < (k, l)$ is in C_m. By the inductive hypothesis, BUILD(S_m, C_m) produces a tree that satisfies $(i, j) < (k, l)$, so the final tree satisfies that constraint also.

Case 2. If k and l are in different blocks, then (k, l) is the root of T. By rule (1) in the definition of π_C, i and j are in the same block, and therefore (i, j) is not the root. Thus $(i, j) < (k, l)$ is surely satisfied. □

```
procedure BUILD(S, C);
if S consists of a single node i then
    return the tree consisting of node i alone
else
    begin
        compute π_C = S_1, S_2, ···, S_r;
        if r = 1 then
            return the null tree
        else
            for m := 1 to r do
                begin
                    C_m := {(i, j) < (k, l) in C | i,j,k,l are in S_m};
                    T_m := BUILD(S_m, C_m);
                    if T_m = the null tree then
                        return the null tree
                end;
        /* if we reach here a tree exists */
        let T be the tree with a new node for its root and
            whose children are the roots of T_m, 1 ≦ m ≦ r;
        return T
    end
end BUILD
```

FIG. 2. *The procedure* BUILD.

LEMMA 1. *Let T be any tree satisfying constraint set C. Then each block of π_C is wholly contained within a subtree rooted at some child of the root of T.*

Proof. Let us consider any fixed order in which we may apply rules (1) and (2) to construct π_C, starting with each leaf in a block by itself. We shall show, by induction on the number of applications of the rules, that each block of the partition being formed always lies wholly within the subtree of some child of the root of T.

Basis. Zero applications. Each leaf is in a block by itself, so the result is trivial.

Induction. Case 1. Rule (1) is applied to a constraint $(i, j) < (k, l)$ causing the blocks containing i and j, say B_1 and B_2, to be merged. By induction, all the nodes in B_1 and B_2 are contained, respectively, in the subtrees rooted at n_1 and n_2, two children of the root. If $n_1 \neq n_2$, then (i, j) is the root of T. But then no constraint of the form $(i, j) < (k, l)$ can be satisfied by T, violating our assumption that T satisfied C. We must therefore have $n_1 = n_2$ and all the nodes in $B_1 \cup B_2$ are descendants of the same child of the root.

Case 2. Rule (2) is applied to constraint $(i, j) < (k, l)$ because k and l are in the same block. Let B_1 and B_2 be the blocks containing i and j, and let B_3 be the block containing k and l. By induction, there exist nodes n_1, n_2 and n_3, children of the root of T, that are the roots of subtrees containing all the nodes of B_1, B_2 and B_3 respectively. Clearly, $n_1 = n_2$, for otherwise (i, j) is the root of T and cannot possibly be a proper descendant of (k, l) as required by the constraint. If $n_1 \neq n_3$, then (i, j) and (k, l) are descendants of different children of the root, again violating the constraint. We must therefore have $n_1 = n_2 = n_3$ and all the nodes in $B_1 \cup B_2 \cup B_3$ are descendants of the same child of the root. □

THEOREM 2. *If there is a tree satisfying the constraint set C, then* BUILD(S, C) *returns some nonnull tree.*

Proof. Suppose not, and assume that T with the constraint set C is as small a counterexample as there is. Suppose BUILD(S, C) returns the null tree. Then either π_C has only one block, or BUILD(S_m, C_m) returns the null tree for some m. In the first case, it follows from Lemma 1 that the root of T has only one child. But then T', which is T with the root removed, provides a smaller counterexample to the theorem.

In the second case, where BUILD(S_m, C_m) returns the null tree, let S_m be contained in the subtree rooted at child n of the root of T. Define T' to be the subtree of T with root n, after deleting all leaves not in S_m and any interior nodes none of whose descendant leaves are in S_m. Then T' satisfies C_m, T' is smaller than T, yet BUILD(S_m, C_m) does not construct a nonnull tree. Thus T together with C was not the smallest counterexample to the theorem, as supposed.

As a single node cannot be counterexample to the theorem, we conclude that the theorem has no smallest counterexample. Since a counterexample, if one exists, is finite, there can be no counterexample at all, from which we conclude the theorem. □

Example 3. Suppose we are given the constraints

$$\begin{array}{ll} (1,3)<(2,5) & (4,5)<(1,9) \\ (1,4)<(3,7) & (7,8)<(2,10) \\ (2,6)<(4,8) & (7,8)<(7,10) \\ (3,4)<(2,6) & (8,10)<(5,9). \end{array}$$

Rule (1) produces partition $\{1, 3, 4, 5\}, \{2, 6\}, \{7, 8, 10\}, \{9\}$. By Rule (2), we must merge the first two blocks because of constraint $(3, 4)<(2, 6)$. Thus the tree constructed top down begins as in Fig. 3(a). The constraints germane to $\{1, 2, \cdots, 6\}$ are $(1, 3)<(2, 5)$ and $(3, 4)<(2, 6)$, while only $(7, 8)<(7, 10)$ is germane to $\{7, 8, 10\}$. The partition for the former is $\{1, 3, 4\}, \{2\}, \{5\}, \{6\}$, and for the latter $\{7, 8\}, \{10\}$. The second level of tree construction is shown in Fig. 3(b). No constraints are applicable to any of the remaining blocks, so at the next level, each block of more than one leaf is partitioned into singletons. The final tree is shown in Fig. 3(c). □

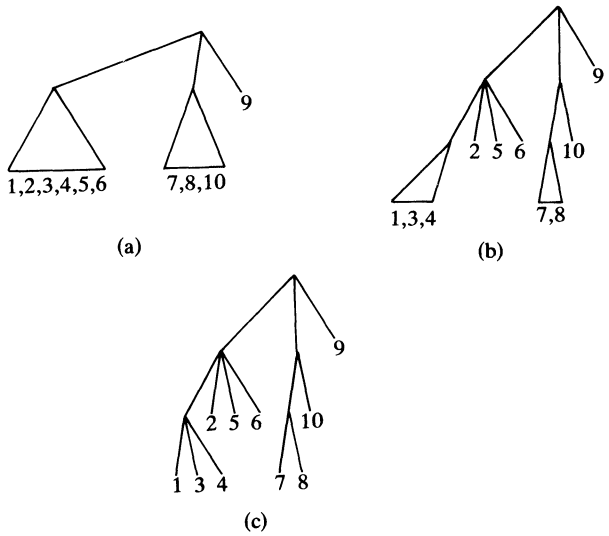

FIG. 3. *Construction of a tree.*

3. Complexity of the tree synthesis algorithm. The running time of the tree synthesis algorithm of § 2 is highly dependent on the method used to partition the set of constraints. In this section, we shall first analyze the running time of BUILD as a

function of n, the number of constraints, and f, a function specifying the time needed to partition a set of constraints. We shall see that the use of the naive partitioning algorithm leads to an $O(n^3)$ implementation of BUILD. By using a more sophisticated partitioning algorithm, the overall running time can be reduced to $O(n^2 \log n)$. Moreover, by imposing the restriction that all constraints be of the form $(i, j) < (i, k)$, we can further reduce the running time to $O(n^2)$. Section 4 of this paper presents a database application in which problems of this restricted form arise naturally.

LEMMA 2. *Let $f(n)$ be the time needed to partition a set of n constraints subject to rules* (1) *and* (2) *of the previous section. If f is monotonically nondecreasing, then the time consumed by* BUILD *when applied to a set of n constraints is $O(nf(n))$.*

Proof. The worst-case running time occurs when the algorithm succeeds in synthesizing a nonnull tree, so we shall restrict our attention to analyzing this case. We shall proceed by assigning a cost to each node of the tree and summing the costs.

Since the top level call on BUILD involves n constraints, each of which can introduce at most $4n$ leaves into the tree, the synthesized tree has at most $4n$ leaves. (Leaves not mentioned in any constraint can be made children of the root of the final tree.) Because the branching factor at each internal node is at least 2, the number of internal nodes cannot exceed $4n - 1$.

At each internal node, BUILD partitions some subset of the original constraints and then spends a constant amount of time (exclusive of recursive calls) processing each block of the partition. Since f is monotonic, $f(n)$ is certainly an upper bound on the cost of performing each partition. Moreover, since f must grow at least linearly with its argument, we can neglect the cost of distributing the constraints over the blocks of the partition when compared with $f(n)$ and charge each internal node $O(f(n))$ units of time. Thus the time charged to all internal nodes is $O((4n-1)f(n)) = O(nf(n))$. Since BUILD spends a constant amount of time at each leaf, the contribution of the leaves to the running time is $O(n)$ for a total over all nodes of $O(nf(n))$. □

The problem of producing a fast implementation of BUILD thus reduces to one of producing a fast partitioning algorithm. Let us first address this problem for the special case mentioned earlier.

THEOREM 3. *If all constraints are of the form $(i, j) < (i, k)$, then* BUILD *can be implemented to run in $O(n^2)$ time, where n is the number of constraints.*

Proof. By Lemma 2, it suffices to show how to partition a set of n constraints in $O(n)$ time. When all constraints are of the form $(i, j) < (i, k)$, rule (2) of the partition constructing algorithm of § 2 need never be applied explicitly. To see this, suppose that i and k are in the same block. By rule (1), i and j must also be in the same block, and thus by transitivity, i, j, and k are all in the same block. Accordingly, it suffices to implement rule (1) alone.

If we take each leaf appearing in one or more constraints to be a node, and each constraint $(i, j) < (i, k)$ to represent an edge between i and j, then we shall have a graph (actually, a multigraph) whose connected components represent the blocks of the partition we are looking for. Since it is easy to find connected components in time proportional to the number of edges [1], and the graph described above has at most n edges, we conclude that partitioning can be done in $O(n)$ time. □

THEOREM 4. *If no restrictions are placed on the form of the constraints, then* BUILD *can be implemented to run in $O(n^2 \log n)$ time.*

Proof. By Lemma 2, it suffices to show how to partition a set of n constraints in $O(n \log n)$ time. Blocks of the partition will be maintained using a set merging algorithm that supports the following operations:

(1) Given an element i, *find* the set that i currently belongs to.

(2) Given two sets, *merge* them together and assign them a common name.

The particular set merging algorithm we need was originally described in [7] and operates as follows. An array S is maintained so that at all times $S[i]$ gives the name of the set currently containing element i. This allows each find operation to be performed in constant time. Each set is represented by a linked list of the current members of the set and an integer specifying the cardinality of the set. Now suppose we want to merge sets i and j. Without loss of generality, let set i have more members than set j. Change $S[k]$ to i for each element k in set j. Then append the list of elements of set j to the list of elements for set i. Finally, update the length of the new set i. It is easy to see that a merge operation can be performed in time proportional to the size of the smaller of the two sets being merged. Thus, starting with a collection of singleton sets, a sequence of n merges can be performed in time $O(n \log n)$.

Returning to the constraint partitioning algorithm, we can use the rules defining the partition π_C to transform a given set of constraints into a set of *commands* of the form $i \equiv j$, which means that i and j must appear in the same block, and a set of *implications* of the form $p \equiv q \Rightarrow r \equiv s$, which means that if p is in the same block as q, then r must be in the same block as s. In the algorithm, each block B of the partition will be associated with a list L_B consisting of all those implications of the form $p \equiv q \Rightarrow r \equiv s$ for which either p or q is currently in B. We assume that the length of each list is stored with it and is available in constant time. We shall also employ a queue Q of commands that remain to be processed. A command is processed by checking that the nodes involved are in the same block of the partition. If not, a merge operation is performed on the appropriate blocks and the algorithm then examines (by traversing the appropriate list L_B) all implications, whose left side involves nodes in the blocks being merged. When an implication is examined, it can cause another command (i.e., its right side) to be generated and placed on Q. The algorithm is shown in Fig. 4.

Before proceeding, notice that each constraint gives rise to one command, which is immediately placed on Q in step (2), and two copies of an implication, say $p \equiv q \Rightarrow r \equiv s$, which are placed on different lists. At most one of these copies will eventually cause the command $r \equiv s$ to be placed on Q, and this will happen when a merge operation first causes p and q to become elements of the same block. Thus the total number of commands enqueued is at most $2n$ (n in step (2) and n in step (3)).

The time spent in steps (1) and (2) is clearly $O(n)$. The time expended by the inner **for** loop of Step (3) is $O(n \log n)$ because each copy of an implication is only considered in this loop when it is being moved from a shorter to a longer list. Since no more than $n-1$ merges can be performed, no implication can be moved more than $\log_2 n - 1$ times. Thus the amount of work done in the inner **for** loop is $O(n \log n)$. The time spent in the rest of step (3), exclusive of the inner **for** loop, is expended in removing at most $2n$ commands from Q, doing at most $O(n)$ finds and comparisons, and performing at most $n-1$ merges. This clearly requires $O(n \log n)$ time, and so the total time used by the algorithm in $O(n \log n)$. □

Extensions. The tree synthesis algorithm of §§ 2 and 3 can be extended to handle a more general set of constraints than the ones considered so far. More specifically, we can handle any collection of constraints of the following types:

(1) $(i, j) < (k, l)$,
(2) $(i, j) \leqq (k, l)$,
(3) $(i, j) = (k, l)$
(4) (i, j) *is comparable to* (k, l),
(5) (i, j) *is incomparable to* (k, l).

In the above constraints, two nodes of a tree are said to be *comparable* if one is the ancestor of the other, and *incomparable* otherwise.

```
(1) for each leaf l mentioned in a constraint do
        begin
            set L_l to the empty list;
            set S[l] to l;
        end;
(2) for each constraint (i, j) < (k, l) do
        begin
            let c be the implication k ≡ l ⇒ i ≡ l;
            add c to L_S[k];
            add c to L_S[l];
            add the command i ≡ j to Q;
        end;
(3) while Q is not empty do
        begin
            remove a command p ≡ q from Q;
            if S[p] ≠ S[q] then
                begin
                    let L be the shorter of L_S[p] and L_S[q];
                    for each implication u ≡ v ⇒ x ≡ y on L do
                        if one of u and v is in S[p]
                            and the other is in S[q] then
                            add the command x ≡ y to Q;
                    append L_S[p] to L_S[q];
                    merge S[p] and S[q];
                end;
        end;
```

FIG. 4. *The general case partitioning algorithm.*

As before, there exists a set of rules for partitioning the leaves of the tree into disjoint sets so that the leaves in each set are descendants of a different child of the root. The appropriate set of rules are:

(1) If $(i, j) < (k, l)$ is a constraint, then i and j are in the same block of π_C.

(2) If $(i, j) < (k, l)$ is a constraint, and k and l are in the same block, then i, j, k and l are all in the same block.

(3) If $(i, j) \leqq (k, l)$ is a constraint, and k and l are in the same block, then i, j, k and l all in the same block.

(4) If $(i, j) = (k, l)$ is a constraint, and i and j are in the same block (or k and l are in the same block), then i, j, k and l are all in the same block.

(5) If (i, j) *is comparable to* (k, l) is a constraint, i and j are all in the same block, and k and l are in the same block, then i, j, k and l are all in the same block.

(6) If (i, j) *is incomparable to* (k, l) is a constraint, then i is in the same block as j, and k is in the same block as l.

(7) No two leaves are in the same block of π_C unless it follows from (1) through (6).

These rules can shown to be both necessary and sufficient conditions for constructing a tree obeying a set of constraints (if one exists) using the procedure BUILD. Each rule involves adding one case to each of Theorem 1 and Lemma 1. A straightforward modification of the partitioning algorithm in § 3 allows us to handle this wider class of constraints without increasing the running time claimed in Theorem 3. For a further generalization of this problem, see [6].

4. An application to relational database queries. In [5], Codd introduced relational algebra as a notation for expressing database queries. In [2], [3] a class of relational expressions called SPJ-expressions was investigated in which the operands of an expression are relations and the operators are the relational algebra operations select, project and natural join.

In [3] it was shown that the value of an SPJ-expression can be represented in terms of a two-dimensional matrix called a *tableau*. A join-minimization procedure for SPJ-expressions was outlined in which a tableau is constructed from a given SPJ-expression E. The tableau is then transformed into an equivalent minimum row tableau. From this minimum row tableau we can then construct an SPJ-expression that has the fewest joins of any relational expression equivalent to the original expression E.

Unfortunately, this optimization procedure may be computationally expensive in that it is NP-complete to minimize the number of rows in a tableau. However, for the important special case of SPJ-expressions with simple tableaux this optimization method can be carried out efficiently. A simple tableau can be minimized in polynomial time [3]. In the remainder of this paper we shall complete the details of the optimization procedure by showing how to use the tree discovery algorithm of § 2 to construct an SPJ-expression from a simple tableau in polynomial time. By way of contrast, Yannakakis and Papadimitriou have shown that it is NP-complete to determine whether an arbitrary tableau has an equivalent SPJ-expression [11].

SPJ-expressions. We assume a data base consisting of a set of two-dimensional tables called *relations*. The columns of a table are labeled by distinct *attributes* and the entries in each column are drawn from a fixed *domain* for that column's attribute. The ordering of the columns of a table is unimportant. Each row of a table is a mapping from the table's attributes to their respective domains. A row is often called a *tuple* or *record*. If r is a relation that is defined on a set of attributes that includes A, and if μ is a tuple of r, then $\mu(A)$ is the value of the A-component of μ.

A *relation scheme* is the set of attributes labeling the columns of a table. When there is no ambiguity, we shall use the relation scheme itself as the name of the table. A relation is just the "current value" of a relation scheme. The relation is said to be *defined on* the set of attributes of the relation scheme. The operators select, project and join are defined as follows.

(1) *Select.* Let r be a relation on a set of attributes X, A an attribute in X, and c a value from the domain of A. Then the *selection* $A = c$ applied to r, written $\sigma_{A=c}(r)$, is the subset of r having value c for attribute A.

(2) *Project.* Let r be a relation on a set of attributes X. Let Y be a subset of X. We define $\pi_Y(r)$, the *projection* of r onto Y, to be the relation obtained by removing all the components of the tuples of r that do not belong to Y and removing duplicate tuples.

(3) *Join.* The join operator, denoted by $\bowtie$, permits two relations to be combined into a single relation whose attributes are the union of the attributes of the two argument relations. Let R_1 and R_2 be two relation schemes with values r_1 and r_2. Then

$$r_1 \bowtie r_2 = \{\mu \mid \mu \text{ is a tuple defined on the attributes in } R_1 \cup R_2, \text{ and there exist tuples } v_1 \text{ in } r_1 \text{ and } v_2 \text{ in } r_2, \text{ such that } v_1(A) = \mu(A) \text{ for all } A \text{ in } R_1 \text{ and } v_2(A) = \mu(A) \text{ for all } A \text{ in } R_2\}.$$

An SPJ-*expression* is an expression in which the operands are relation schemes and the operators are select, project and join. An SPJ-expression with operands $R_1, R_2, \cdots, R_n$ evaluates to a single relation when relations are assigned to $R_1, R_2, \cdots, R_n$.

Definition of a tableau. A tableau is a matrix in which the columns correspond to the attributes of the universe in a fixed order. The first row of the matrix is called the *summary* of the tableau. The remaining rows are exclusively called *rows*. The general idea is that a tableau is a shorthand for a set former $\{a_1 \cdots a_n | \psi(a_1, \cdots, a_n)\}$ that defines the value of a relational expression. In the set former ψ does not have any a_i's as bound variables. To simplify later discussion we shall adopt the following conventions regarding tableaux. The symbols appearing in a tableau are chosen from:

(1) Distinguished variables, for which we use a's, possibly with subscripts. These correspond to the symbols to the left of the bar in ψ.

(2) Nondistinguished variables, for which we generally use b's. These are the bound variables appearing in the set former.

(3) Constants, for which we use c's or nonnegative integers.

(4) Blank.

The summary of a tableau may contain only distinguished variables, constants, and blanks. The rows of a tableau may contain variables (distinguished and nondistinguished) and constants. When tableaux represent SPJ-expressions, we can assume that the same variable does not appear in two different columns of a tableau, and that a distinguished variable does not appear in a column unless it also appears in the summary of that column.

Let T be a tableau and let S be the set of all symbols appearing in T (i.e., variables and constants). A *valuation* ρ for T associates with each symbol of S a constant, such that if c is a constant in S, then $\rho(c) = c$. We extend ρ to the summary and rows of T as follows. Let w_0 be the summary of T, and $w_1, w_2, \cdots, w_n$ the rows. Then $\rho(w_i)$ is the tuple obtained by substituting $\rho(v)$ for every variable v that appears in w_i.

A tableau defines a mapping from *universal instances* (relations over the set of all attributes) to relations over a certain subset of the attributes, called the *target relation scheme*, in the following way. If T is a tableau and I an instance, then $T(I)$ is the relation on the attributes whose columns are nonblank in the summary, such that

$$T(I) = \{\rho(w_0) | \text{for some valuation } \rho \text{ we have } \rho(w_i) \text{ in } I \text{ for } 1 \leqq i \leqq n\}.$$

Example 4. Let T be the tableau

A	B	C
a_1	a_2	
a_1	b_1	b_3
b_2	a_2	1
b_2	b_1	b_4

We conventionally show the summary first, with a line below it. We can interpret this tableau as defining the relation on AB

$$T(I) = \{a_1 a_2 | (\exists b_1)(\exists b_2)(\exists b_3)(\exists b_4) \text{ such that } a_1 b_1 b_3 \text{ is in } I \text{ and } b_2 a_2 1 \text{ is in } I \text{ and } b_2 b_1 b_4 \text{ is in } I\}$$

given any universal instance I.

Equivalence. Two SPJ-expressions $E_1(R_1, \cdots, R_m)$ and $E_2(S_1, \cdots, S_n)$ are said to be *equivalent* if, for all universal instances I, $E_1(r_1, \cdots, r_m) = E_2(s_1, \cdots, s_n)$, where $r_i = \pi_{R_i}(I)$, $1 \leqq i \leqq m$, and $s_i = \pi_{S_i}(I)$, $1 \leqq i \leqq n$. In words, E_1 is equivalent to E_2 if they represent the same mapping on universal instances.

Similarly, two tableaux T_1 and T_2 are equivalent if, for all I, $T_1(I) = T_2(I)$. Likewise, a tableau T is equivalent to an expression $E(R_1, \cdots, R_n)$ if, for all I, $T(I) = E(r_1, \cdots, r_n)$ where $r_i = \pi_{R_i}(I)$ for $1 \leqq i \leqq n$.

Representation of SPJ-expressions by tableaux. Given an SPJ-expression E, we can construct a tableau T to represent the expression in the following manner. The construction proceeds inductively on the form of E.

(1) If E is a single relation scheme R, then the tableau T for E has one row and a summary such that:
  (i) If A is an attribute in R, then in the column for A tableau T has the same distinguished variable in the summary and row.
  (ii) If A is not in R, then its column has a blank in the summary and a new nondistinguished variable in the row.

(2a) Suppose E is of the form $\sigma_{A=c}(E_1)$, and we have constructed T_1, the tableau for E_1.
  (i) If the summary for E_1 has blank in the column for A, then $T = T_\varnothing$, where $T_\varnothing$ is a tableau that maps any universal instance into the empty set.
  (ii) If there is a constant $c' \neq c$ in the summary column for A, then $T = T_\varnothing$. If $c = c'$, then $T = T_1$.
  (iii) If T_1 has a distinguished variable a in the summary column for A, the tableau T for E is constructed by replacing a by c wherever it appears in T_1.

(2b) Suppose E is of the form $\pi_X(E_1)$, and T_1 is the tableau for E_1. The tableau T for E is constructed by replacing nonblank symbols by blanks in the summary of T_1 for those columns whose attributes are not in X. Distinguished variables in the rows of those columns are consistently replaced with new nondistinguished variables.

(2c) Suppose E is of the form $E_1 \bowtie E_2$ and T_1 and T_2 are the tableaux for E_1 and E_2, respectively. Let S_1 and S_2 be the symbols of T_1 and T_2, respectively. Without loss of generality, we may take S_1 and S_2 to have disjoint sets of nondistinguished variables, but identical distinguished variables in corresponding columns.
  (i) If T_1 and T_2 have some column in which their summaries have distinct constants, then $T = T_\varnothing$.
  (ii) If no corresponding positions in the summaries have distinct constants, the set of rows of the tableau T for E consists of the union of all the rows of T_1 and T_2. The summary of T has in a given column:
    (a) The constant c if one or both of T_1 and T_2 have c in that column's summary. In this case we also replace any distinguished variable in that column by c.
    (b) The distinguished variable a if (a) does not apply, but one or both of T_1 and T_2 have a in that column's summary.
    (c) Blank, otherwise.

It is not hard to show that the tableau constructed by this procedure is equivalent to the given expression. Note that the number of rows in the resulting tableau is one more than the number of join operators in the original expression.

Example 5. Let A, B and C be the attributes, in that order, and suppose we are given the expression $\pi_A(\sigma_{B=0}(AB \bowtie BC))$. By Rule (1), the tableaux for AB and BC are

A	B	C
a_1	a_2	
a_1	a_2	b_1

and

A	B	C
	a_2	a_3
b_2	a_2	a_3

By Rule (2c), the tableau for $AB \bowtie BC$ is

A	B	C
a_1	a_2	a_3
a_1	a_2	b_1
b_2	a_2	a_3

By Rule (2a), the tableau for $\sigma_{B=0}(AB \bowtie BC)$ is

A	B	C
a_1	0	a_3
a_1	0	b_1
b_2	0	a_3

Finally, by Rule (2b), the tableau for $\pi_A(\sigma_{B=0}(AB \bowtie BC))$ is

A	B	C
a_1		
a_1	0	b_1
b_2	0	b_3

Simple tableaux. A tableau is *simple* if in any column with a repeated nondistinguished variable there is no other symbol that appears in more than one row. For simple tableaux there exists a polynomial-time equivalence algorithm, whereas for general tableaux the equivalence problem is NP-complete [3]. In practice, it is not easy to find an SPJ-expression with a nonsimple tableau. The expression $\pi_{AC}(AB \bowtie BC) \bowtie (AB \bowtie BD)$ is a minimal expression that gives rise to a nonsimple tableau. The tableau is shown in Fig. 5. The rows in the column for B have repeated nondistinguished and distinguished variables.

A	B	C	D
a_1	a_2	a_3	a_4
a_1	b_1	b_2	b_3
b_4	b_1	a_3	b_5
a_1	a_2	b_6	b_7
b_8	a_2	b_9	a_4

FIG. 5. *A nonsimple tableau.*

5. Synthesis of relational expressions from tableaux. In this section we shall present an algorithm for constructing a relational expression from a simple tableau. To help clarify the presentation, we shall portray a relational expression by its parse tree.

Although many relational expressions can be synthesized from the same tableau, the relational expression produced by our algorithm has a parse tree with the following properties:

(1) Each project operation is done as soon as possible (i.e., is as low in the parse tree as possible).

(2) Each select operation is applied to a leaf.

These two points are motivated by efficiency considerations; performing projections and selections as early as possible can significantly reduce the size of intermediate relations computed in the evaluation of a relational expression. We should point out, however, that to evaluate a relational expression efficiently in practice, one must take into account many parameters of the database environment such as the costs of the various data access methods that are available and the nature of the data structures used to store the relations. See [8], [9], [10], [12] for more discussion of query evaluation strategies.

We shall assume the leaves of a parse tree are labeled by relation schemes. In the absence of other information, we can choose as leaf labels relation schemes having as few attributes as possible. For example, the expressions $\pi_A(A)$, $\pi_A(AB)$, $\pi_A(AC)$, and $\pi_A(ABC)$ all have the same tableau. Of these expressions, $\pi_A(A)$ is minimal in that the other expressions can be produced from it by simply adding one or more attributes to the relation scheme which is the operand of the expression.

For the application considered here, however, the problem of deciding which relation scheme corresponds to which leaf of the tree vanishes. Here we are interested in minimizing the number of joins in a given SPJ-expression. One way to perform this optimization is to construct a tableau for the given expression, minimize the number of rows in the tableau using the procedure in [2], and then convert the resulting tableau back to an SPJ-expression. The number of rows in the tableau is one more than the number of joins in the associated expression. In the process, it is easy to keep track of which rows correspond to which operands of the original expression. Since the tableau minimization algorithm of [2] can only delete rows but never change one, we can associate each leaf of the tree eventually produced with a relation scheme appearing in the original expression. Accordingly, we shall henceforth assume that each row of the tableaux under consideration is identified with some given relation scheme.

Let us initially consider a simple tableau T with no constants in the summary, although constants may appear in the rows. Let A be a column in which rows i and j have the same nondistinguished variable, and k a row with a distinguished variable in column A[1]. If T comes from an expression, consider the parse tree P of that expression. Each leaf of P corresponds to an operand relation scheme associated with some row of the tableau.

Suppose that in P (i, j) is not a proper descendant of (i, k). Since (i, j) and (i, k) cannot be independent, it follows that (i, k) must be a descendant of (i, j). Then consider the tableau constructed for the subexpression rooted at the node (i, k) of P. In particular, which rows have the distinguished variable in column A? Surely k does, as the distinguished variable appears there in tableau T, and in the tableau construction algorithm a nondistinguished variable is never changed into a distinguished variable. If row i does not also have the distinguished variable in column A, then the variables in

[1] In a simple tableau, k will be unique, but we can relax the "simple" constraint to the point that a column with a repeated nondistinguished variable can have a repeated distinguished variable or constant, but not another repeated nondistinguished variable.

rows i and j cannot be equated when we create the tableau for the expression represented by node (i, j). However, if both rows i and k have the distinguished variable, then they must have the same variable in T, a contradiction. We can therefore infer that $(i, j) < (i, k)$. Arguing similarly, we can show that $(i, j) < (j, k)$. From these two observations we are led to the conclusion:

(*) Suppose P is the parse tree of an expression yielding tableau T. If rows i and j of T have the same nondistinguished variable in column A, and row k has a distinguished variable in column A, then we must have $(i, j) < (i, k)$ and hence $(i, j) < (j, k)$ in P.

That this is sufficient for simple tableaux is expressed in the next theorem.

THEOREM 5. *Let T be a tableau having no constants in the summary and no column with two or more repeated nondistinguished variables. Then T comes from an expression if and only if the set of constraints defined in* (*) *determines a tree.*

Proof. Only if. We argued above that if T comes from an expression, the constraints (*) must be satisfied.

If. Using the algorithm of § 2, suppose (*) determines a tree P. We shall show how to convert this tree into an expression for T. This expression may use join as an n-ary operator. But since binary join is associative and commutative, nothing is lost by doing so, and the n-ary joins can be replaced by binary joins in any order.

The construction proceeds by induction on the height of a node in the tree P. For the inductive hypothesis we shall construct for each subtree an expression yielding a tableau whose rows are the rows of T that are leaves of the subtree, but with repeated nondistinguished variables, all of whose occurrences do not appear among these rows, replaced by distinguished variables. The conditions on T guarantee that no conflicts arise, that is, no column needs more than one distinguished variable.

Basis. Height 0. The node is a leaf labeled by a relation scheme R, and identified with some row i. Row i cannot have a constant or distinguished variable in columns other than those for attributes in R. (We assume the relation scheme R containing all attributes in which row i has a distinguished variable, constant, or repeated nondistinguished variable is a legitimate operand. If not all relation schemes are permissible, then it is easy to check for tableaux with illegal rows.)

If row i has constant c in column A, then apply $\sigma_{A=c}$ to R. Then project onto those attributes in whose column row i has either a distinguished variable or a repeated nondistinguished variable. The result is an expression whose tableau has one row, which is row i with repeated nondistinguished variables replaced by distinguished variables.

Induction. We construct the expression for node n from the expressions $E_1, E_2, \cdots, E_m$ representing the children of node n. Begin by joining $E_1, E_2, \cdots, E_m$. Then project onto those attributes A in whose columns either

(1) some descendant leaf of n is a row with a distinguished variable in column A, or
(2) some, but not all, of the occurrences of the repeated nondistinguished variable in column A are found in rows which are descendants of n.

The result is easily seen to satisfy the inductive hypothesis, as in each column A, variables that appear in two or more of the E_i's and that must become the repeated nondistinguished variable for column A are distinguished in the tableaux for the E_i's and are therefore equated in the join.

Finally, the inductive hypothesis applied to the root implies that the constructed expression has tableau T. □

The proof of Theorem 5 contains the algorithm to construct an expression from a tableau. The following example illustrates the procedure.

Example 6. Consider the tableau

	A	B	C	D
		a_1	a_2	a_3
1	0	b_1	b_2	b_3
2	b_4	b_1	a_2	b_5
3	b_6	b_7	a_2	a_3
4	b_8	a_1	a_2	b_9

We assume that rows 1, 2, 3 and 4 come from relation schemes (AB), (BC), (CD) and (BC), respectively. Since b_1 is the only repeated nondistinguished variable in the tableau, the only constraints are $(1, 2) < (1, 4)$ and $(1, 2) < (2, 4)$. Applying the procedure BUILD to these constraints we obtain the tree shown in Fig. 6.

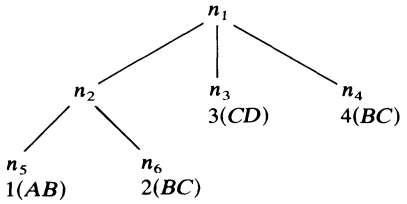

FIG. 6. *Initial tree.*

Using the construction in the basis for node n_5 we obtain the expression $\pi_B(\sigma_{A=0}(AB))$, while for the leaves n_6, n_3 and n_4, the expressions (BC), (CD) and (BC) suffice. Using the construction in the inductive step for node n_2 we obtain the expression $\pi_C(\pi_B(\sigma_{A=0}(AB))\bowtie(BC))$. We project onto column C because only column C has a distinguished variable in rows 1 or 2, and all occurrences of the repeated nondistinguished variable b_1 are found in these rows. The expression for n_1 is obtained by joining the above expression with (CD) and (BC) in any order. We should then project onto BCD, but this projection is seen to be superfluous, since the final join produces a relation over only those attributes. The parse tree for the final expression is shown in Fig. 7. □

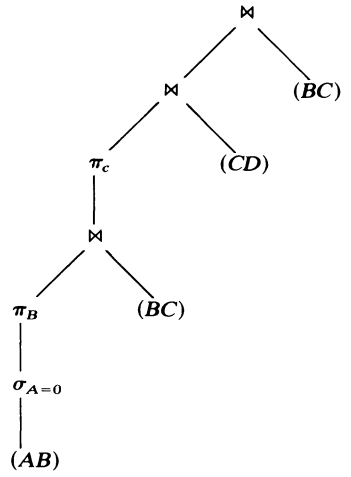

FIG. 7. *Synthesized expression.*

Extension to the case where constants appear in the summary. If there is a constant c in column A of the summary of a tableau, then c must appear in some row of column A, else the tableau can be shown not to come from an expression. If c appears in only one row k of column A, we may treat c as a distinguished variable, introducing the constraints $(i, j) < (i, k)$ and $(i, j) < (j, k)$ for each pair of rows i and j having a repeated nondistinguished variable in column A.

If c appears in more than one row of column A, we have choices. For all but one of these rows we can select $A = c$ and then project A out. But for one row we must treat c as if it were distinguished, and introduce the appropriate constraints. Since we may have a choice for each column that has a constant in the summary, we apparently have a combinatorial problem. However, we may adopt the simple expedient of permitting a new operator *augment*, defined by $\alpha_{A=c}(r) = \{\mu \mid \mu(A) = c$ and there exists v in r such that for all attributes B on which r is defined, $\mu(B) = v(B)\}$. Note that we may assume that r is not defined on A; otherwise augment is the same as select. Then we may synthesize an expression from a tableau with constants in the summary by:

(1) deleting constants from the summary,

(2) synthesizing an expression for the resulting tableau, if it exists, and then

(3) using the augment operator to introduce the constants into the tuples of the relation resulting from application of the expression from (2).

6. Finding a minimal expression equivalent to a given tableau. We should also consider a variant of the problem of finding an expression that yields a given tableau. In most circumstances it will be sufficient to find an expression that yields an equivalent tableau, that is, an expression defining the same mapping from universal instances to target relations as the tableau.

It turns out that this question is no harder than the original problem, as we can show that a tableau comes from an expression only if its minimal equivalent tableau does. Thus, if we minimize the number of rows in a tableau, then we can obtain an expression with the fewest joins equivalent to a given tableau since the number of rows in the tableau is one more than the number of joins in the resulting expression.

THEOREM 6. *If a tableau comes from an expression, then its minimal equivalent tableau comes from an expression.*

Proof. It follows from [4] that any given tableau T has a minimal equivalent tableau T' (one with the fewest number of rows) such that each row of T' is a row of T. Given an expression E yielding tableau T, we can delete the operands of E corresponding to rows of T that are not in T'.

Nodes that apply unary operators (select and project) to a deleted operand, and nodes that join two deleted operands are themselves deleted. Nodes that join a deleted operand with one that is not deleted are identified with the nondeleted operand. The resulting expression will yield tableau T'. We may prove by an easy induction on the

	A	B	C	D
	a_1	a_2	a_3	
1	a_1	b_1	b_2	b_3
2	b_4	b_1	b_5	b_6
3	a_1	a_2	b_7	b_6
4	b_8	a_2	a_3	b_9

FIG. 8. *A tableau.*

height of a node remaining in the expression that the tableau for this node is the same as the tableau for the corresponding node in E, with rows that do not eventually become rows in T' deleted. □

Example 7. The tableau in Fig. 8 implies constraints $(1, 2) < (1, 3)$, $(1, 2) < (2, 3)$, $(1, 2) < (1, 4)$, and $(1, 2) < (2, 4)$, so we may synthesize the expression of Fig. 9. The minimal equivalent tableau for Fig. 8 has only rows 3 and 4. To synthesize an expression for the minimal tableau, we delete nodes n_8 and n_9, which correspond to rows 1 and 2. This causes nodes n_7 and n_5 to be deleted and node n_4 to be merged with n_6. The resulting expression is shown in Fig. 10.

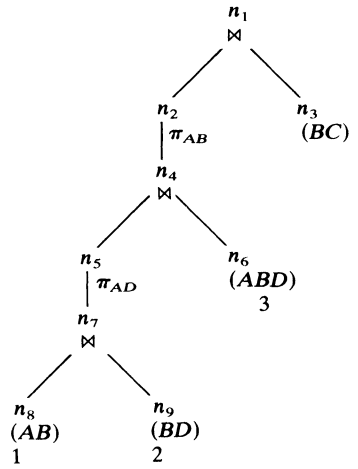

FIG. 9. *An expression for the tableau of Fig.* 8.

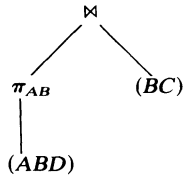

FIG. 10. *Expression for minimal tableau.*

7. Summary. In this paper we have presented an $O(n^2 \log n)$ algorithm that will construct, whenever possible, a tree to satisfy n given constraints on common ancestors of a set of nodes. The constraints specify lineage restrictions on lowest common ancestors of pairs of nodes. We have also shown that the tree construction algorithm can be used as part of a process to minimize the number of joins in a certain class of relational algebra expressions containing only select, project and join operators.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] A. V. AHO, Y. SAGIV AND J. D. ULLMAN, *Efficient optimization of a class of relational expressions*, ACM Trans. Database Systems, 4 (1979), pp. 435–454.

[3] ———*Equivalences among relational expressions*, this Journal, 8 (1979), pp. 218–246.

[4] A. K. CHANDRA AND P. M. MERLIN, *Optimal implementation of conjunctive queries in relational databases*, Proc. Ninth Annual ACM Symposium on Theory of Computing 1976, pp. 77–90.
[5] E. F. CODD, *A relational model for large shared data banks*, Comm. ACM, 13 (1970), pp. 377–387.
[6] P. J. DOWNEY, R. SETHI AND R. E. TARJAN, *Variations on the common subexpression problem*, J. Assoc. Comput. Mach., 27 (1980), pp. 758–771.
[7] J. E. HOPCROFT AND J. D. ULLMAN, *Set merging algorithms*, this Journal, 2 (1973), pp. 294–303.
[8] P. G. SELINGER, M. M. ASTRAHAN, D. D. CHAMBERLIN, R. A. LORIE AND T. G. PRICE, *Access path selection in a relational database management system*, Proc. ACM SIGMOD Conference on Management of Data, Boston, MA, 1979, pp. 23–24.
[9] J. D. ULLMAN, *Principles of Database Systems*, Computer Science Press, Potomac, MD, 1980.
[10] E. WONG AND K. YOUSSEFI, *Decomposition—a strategy for query processing*, ACM Trans. Database Systems, 1 (1976), pp. 223–241.
[11] M. YANNAKAKIS AND C. PAPADIMITRIOU, *Algebraic dependencies*, Proc. 21st Annual IEEE Symposium on Foundations of Computer Science, 1980, pp. 328–332.
[12] S. B. YAO, *Optimization of query evaluation algorithms*, ACM Trans. Database Systems, 4 (1979), pp. 133–155.