A Study of Transitive Closure As a Recursion Mechanism

H V Jagadish and Rakesh Agrawal, AT&T Bell Laboratories, Murray Hill, NJ

and Linda Ness, University of Texas at Austin, Austin, TX[1]

ABSTRACT

We show that every linearly recursive query can be expressed as a transitive closure possibly preceded and followed by operations already available in relational algebra This reduction is possible even if there are repeated variables in the recursive literals and if some of the arguments in the recursive literals are constants Such an equivalence has significant theoretical and practical ramifications One the one hand it influences the design of expressive notations to capture recursion as an augmentation of relational query languages On the other hand implementation of deductive databases is impacted in that the design does not have to provide the generality that linear recursion would demand It suffices to study the single problem of transitive closure and to provide an efficient implementation for it

1 Introduction

A major stumbling block in the success of database systems augmented with logic-based query languages has been the efficient implementation of recursion in logic queries although numerous strategies have been proposed to deal with it (see [1 2] for a comprehensive survey) The problem of efficiently implementing general recursion is a hard problem However it has been conjectured that in practice most recursive queries are linear [2] and substantial research has been devoted to the study of linear recursion as a special case (see for example [3-9]) In this paper we show that every linearly recursive query can be expressed as a transitive closure possibly preceded and followed by operations already available in relational algebra An implication of this result is that the designer of a deductive database does not have to deal even with the generality that the linear recursion demands and it is sufficient to study the single problem of transitive closure Transitive closure is a well understood operation, and efficient iterative [4 10 11] and direct [12-17] algorithms exist for its implementation One could therefore build efficient deductive database systems using transitive closure as a primitive recursion operator Our result also has a significant impact on the design of expressive notations to capture recursion as an augmentation of relational algebra (see [18])

The organization of the rest of the paper is as follows In Section 2 we define what we mean by a linear recursion and by transitive closure as applied to database queries In Section 3 we set introduce the concept of a substitution graph and set up the analytical tools necessary to obtain our results Section 4 is the heart of the paper and shows how to convert a linear recursion with no repeated variables in the consequent into a transitive closure form In Section 5, we show how the case of repeated variables in the consequent can also be handled We put it all together and give a complete procedure for evaluating a

1 Most of the results in this paper were discovered independently by Jagadish and Agrawal at Bell Laboratories and by Ness at the University of Texas Support for Ness was provided in part by NSF under contract numbers MCSS 8104017 MCS-8214613, and DCR-8507224, and by ONR under contract number N00014-86-K-0161

linear recursion as a transitive closure in Section 6 along with some hints to obtain efficiency Finally Section 7 we discuss some of the implications of our work

2 Definitions

2.1 Preliminaries

A *literal* is of the form $p(v_1, v_2 \quad v_n)$ where p is a predicate name of arity n and each argument v_i is a constant or a variable A *rule* is a statement of the form

$p - q1\, q2 \quad qm$

where p and the q_i's are literals and can be read as "p is implied by the conjunction of $q1$ $q2$ qm' The literal p is called the *consequent* and the conjunction of the q_is is called the *antecedent* of the rule A rule is *recursive* if for some i $q_i = p$ and p is called the *recursive literal* In a recursive rule, all q_j such that $q_j \neq p$ are called the *nonrecursive literals* A recursive rule is *linear* if there is exactly one occurrence of the recursive literal in the antecedent We make the popular assumption that there are no function symbols in the statement We also assume that each of q_i's is a base literal (that is a relation explicitly stored in the database) We will call the arguments of the consequent *distinguished arguments* The literals may contain repeated arguments, in particular the consequent and recursive literal of a linear recursive rule may contain repeated arguments

We introduce the concept of the *generation* of a tuple that satisfies the recursive predicate A tuple that is given at the start of the recursive evaluation is said to be of generation zero A tuple that can be derived by application of the recursive rule once to a tuple of generation k, is said to be of generation $k+1$ It is possible for a tuple to belong to several different generations if it has several different derivations Recursive evaluation completes at some generation N when all the tuples of generation N or greater also belong to some generation less than N

2.2 The Basic Form for Transitive Closure

A recursive rule is said to be in the *basic form for transitive closure* if it is written as

$$p(X,Z) - p(X,Y)\, q(Y,Z) \qquad (1)$$

If p_0 is the initial value for the recursive literal, then the solution to such a query is known to be

$$p_0(X,Y)q(Y,Z) - p_0(X,Y)q(Y,V)q(V,Z) + p_0(X,Y)q(Y,U)q(U,V)q(V,Z) -$$

This expression can be written more concisely as

$$p_0(X,Y)\, q^*(Y,Z) \qquad (2)$$

where q^* is the (reflexive and transitive) closure of q under composition with itself That is $q^* = q + qoq + qoqoq +$ where o is the composition" [19] operator so that $(qoq)(Y,Z) - q(Y,V)q(V,Z)$ We shall call V the *right composition field* of $q(Y,V)$ and the *left composition field* of $q(V,Z)$

There is no reason why X, Y Z in eqn (1) should be single argument fields These could be *argument tuples* (or vector' fields consisting of several arguments Thus X could represent $v_1, v_2 \quad v_n$ for some n Similarly for Y and Z The tuples Y and Z have an identical number of arguments In the basic form we impose the restriction that the arguments in X Y and Z are all distinct

2.3 The Non-Recursive Literal in the Basic Form

A recursive rule may fail to be in the basic form for transitive closure because the arguments of the non-recursive literal(s) are not those in the argument tuples Y and Z as above For example, a recursive rule may be written as

$$p(X,Z) - p(X,Y)q(X',Y',Z',W)$$

where X' Y' Z' are subsets of the disjoint sets of arguments X Y and Z respectively, and W

consists of arguments which do not appear in the consequent or recursive literal First of all one can immediately project out the W' arguments They do not participate in the closure The arguments in X' remain the same throughout the recursion Therefore one could first perform the indicated selection Y on the relation q and then project out the X arguments also obtaining $q(Y'Z')$ Then the query can be rewritten as -

$$p(XZ) - p(XY)q(Y'Z')$$

Here q' may be viewed as the projection of a relation $q(YZ)$ so the solution may be expressed as in eqn (2)

$$p_0(XY)q'^*(YZ') \qquad (2)$$

Here the joins are taken over the fields that actually occur The other arguments of q that do not occur in q' can take on any arbitrary values and thus trivially participate in the composition The subsets Y' and Z' may even be empty resulting in a trivial closure

The apparent requirement that there is just one non-recursive literal is just a convenience One could always take the cartesian product of multiple non-recursive literals if present and thus obtain a single non-recursive literal Better ways to do this are discusses in section 6 2

2 4 A Canonical Form for Transitive Closure

The form of recursion that actually arises in our procedure is only slightly more general that the basic form for transitive closure One definition is necessary before introducing this form

Consider two tuples $U = (u_1\ u_2 \quad u_n)$ and $V = (v_1\ v_2 \quad v_n)$ U and V are said to be *isomorphic* if any common arguments are in the same positions in both tuples and any noncommon arguments are variables and the pattern of repetition of arguments is the same in both tuples In other words consider one of the arguments of U say u_1 If this argument appears in V it should do so as v_1 and not as any other argument Further if u_1 is equal to some other argument in U say u_5 then v_1 should also be equal to v_5 and either both v_1 and v_5 are equal to u_1 or neither is

Now consider the arguments to the recursive predicate on the two sides of eqn (1) defining the basic form for transitive closure It is easy to see that in the absence of repeated arguments the tuple XZ is isomorphic to the tuple XY (The X arguments are all identical in the two tuples The Y and Z arguments are all distinct so that the pattern of repetition is trivially identical) On the other hand consider any two isomorphic tuples Each tuple can be split into two parts one that consists of arguments that are common to the two tuples and another that consists of arguments private to only one Calling the common part X and the private parts Y and Z we can write the two isomorphic tuples as XY and XZ respectively where we guarantee that there is no argument that is in common between any two of X Y and Z If these isomorphic tuple were to be the arguments of the recursive predicate on either side of a linear recursive rule any repetition of arguments in X or in both Y and Z can trivially be projected out without affecting the recursion One is then left with an X Y and Z with all arguments distinct just as in the basic form for transitive closure

In the light of the above discussion we say that a linear recursive rule

$$p(X) \ - p(W)\ q_1(Z_1)\ q_2(Z_2) \quad q_m(Z_m) \qquad (3)$$

is in the *canonical form* for transitive closure if X is isomorphic to W Notice that the attention is now focused on the argument tuples of the recursive literals

3 Our Constructs

In this section, and in the next section, we assume that there are no repeated arguments in the consequent though there may be repeated

arguments in the antecedent recursive literal and there may be constants in either or both The extension to the case with repeated arguments in the consequent is straightforward and is presented in section 5

3 1 The Substitution Map for a Linear Recursive Rule

We are going to attempt to convert a linear recursive rule into the canonical form for transitive closure Doing so involves rendering the X and Y tuples isomorphic given an equation such as eqn (3)

Obviously the information of interest is how the argument tuples of the recursive literal on either side of the linear recursive rule relate Let us write a linear recursive rule as

$$p(X) - p(S(X\ Y))q(\ ) \tag{4}$$

X denotes the argument tuple for the consequent and is called the *distinguished argument tuple* Y is called the *non-distinguished argument tuple* and represents the arguments in the body of the rule that are not in the head also The arguments to the non-recursive literal(s) are not considered from this point on and are merely represented by a single dot S is called the *substitution map* for the recursive rule in eqn (4) It should be viewed as a (linear) map from tuples to tuples What it basically says is that the argument tuple W of eqn (3) is obtained as a permutation on (a subset of) the argument tuple X and the non-distinguished argument tuple

It is useful to think of the recursion as being evaluated top-down starting from the goal Thus we have X as the argument tuple to the recursive predicate when we start After one application of the recursive rule we get $S(X\ Y_1)$, where Y_1 is the non-distinguished argument tuple or the tuple of new" arguments introduced on the first application of the recursive rule After the second application, the argument tuple to the recursive literal is $S(S(X,Y_1)\ Y_2)$ and so on for successive applications We shall find it convenient to write the tuples so obtained as $S^2(X)$ $S^3(X)$ and so forth with the implicit understanding that new arguments may also be introduced through Y_1 Y_2 etc

Since S is a linear substitution map it can be written as a matrix if the argument tuples are written as vectors Thus we can write

$$\vec{W} = \mathbf{S}\begin{pmatrix}\vec{X}\\ \vec{Y}\end{pmatrix} \tag{5}$$

where $\mathbf{S}$ is a matrix consisting of only zeroes and ones representing the substitution map (Matrix element s_{ij} is 1 if and only if w_i is the same as the j^{th} element in the composite vector of X and Y)

The substitution map S can be viewed as a combination of a substitution map for the distinguished argument tuple (the first several columns of the matrix $\mathbf{S}$ corresponding to X) and a substitution map for the non-distinguished argument tuple (the remaining columns of $\mathbf{S}$) We will denote the substitution map for the distinguished argument tuple by S' and refer to it as the *distinguished substitution map* of the recursive rule It is defined as

$$S'(v_1 \quad v_n)=(v_1 \quad v_n)$$

where

1 $v_i=0$ if no distinguished variable is substituted into the i^{th} argument position in the antecedent recursive literal
2 $v_i=v_j$ if the variable in position i in the consequent appears in position j in the antecedent recursive literal

The matrix for S' is an $n\times n$ matrix which has a 1 in the i^{th} row and j^{th} column if $v_i=v_j$ and has zeros elsewhere It should be evident that the distinguished substitution map after k applications of the recursive rule is simply $S'(S'(S'\ S'(X))\ ))$, which could be written $S'^k(X)$ In matrix form, we could directly write $\mathbf{S}'^k\vec{X}$

Notice that eqn (5) could be rewritten

$$\vec{W} = \mathbf{S}'\vec{X} + \mathbf{U}\vec{Y} \tag{6}$$

where **U** is the matrix corresponding to the non-distinguished substitution map In other words $S(X\,Y) = S'(X) + U(Y)$ Now after two applications of the recursive rule we can write the substitution map as $S(S(X\,Y_1)\,Y_2) = S'(S'(X) + U(Y_1)) + U(Y_2)$ Proceeding in a similar fashion and expanding each time we can write the substitution map obtained after k applications of the recursive rule as

$$S'^{k}(X) + S'^{k-1}(U(Y_1)) + \\ + S'(U(Y_{j-1})) + U(Y_j) \tag{7}$$

3 2 The Substitution Graph for a Linear Recursive Rule

The matrix for the distinguished substitution map **S** is the adjacency matrix for a directed graph which we will call the *substitution graph* This substitution graph for a rule has

1 a node for each distinguished argument position labeled by that position number and
2 an edge from node i to node j if the argument in position i in the consequent is substituted into position j in the antecedent recursive literal

Note that the substitution graph for a recursive rule in the basic form for transitive closure consists of nodes with self-loops and independent nodes The matrix for the substitution transformation has a 1 on the diagonal for each node with a self-loop and has zeroes elsewhere

Example [1] *An example query with a nontrivial substitution graph*

Messages are communicated between transmitters each of which can be in one of a finite number of states (or modes) No transmission is allowed between transmitters which are in the same states The possible communication channels are stored in the relation q A tuple in the relation $q(S_1\,S_2\,T_1\,T_2)$ represents the fact that transmitter T_1 when it is in state S_1 can send a message to transmitter T_2 when it is in state S_2, each tuple in the relation q is called a channel Now suppose that some of the channels are special perhaps they are heavily used or blocked or connect transmitters in different countries These special channels are stored in a relation e We want to query for the channels which are connected to a special channel via a succession of channels in alternating states Messages transmitted over these channels might fail to be transmitted successfully to their destination or might have a higher probability of being intercepted The alternating state requirement is just another constraint on the message transmissions

The query for the relation $p(S_1\,S_2\,T_1\,T_2)$ of channels which lead to special channels through transmitters in alternating states is given below

$$p(S_1\,S_2\,T_1\,T_2) - e(S_1\,S_2\,T_1\,T_2)$$
$$p(S_1\,S_2\,T_1\,T_4) - q(S_1\,S_2\,T_1\,T_2)p(S_2\,S_1\,T_2\,T_3)q(S_1\,S_2\,T_3\,T_4)$$

In this example the distinguished argument tuple X is $(S_1\,S_2\,T_1\,T_4)$ The non-distinguished argument tuple Y is $(T_2\,T_3)$ $S(X\,Y)$ is obviously $(S_2\,S_1\,T_2\,T_3)$ and $S(X\,Y)$ is $(S_2\,S_1\,0\,0)$ In matrix form we can write

$$\mathbf{S}' = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \qquad \mathbf{U} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\mathbf{S} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

The substitution graph is shown below

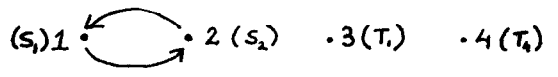

Figure 1 The Substitution Graph for the Transmitter Example

3 3 Multiple Step Recursions

Given an initial set of tuples P_0 satisying the recursive predicate, we obtain the tuples of the first generation P_1 by application of the given recursive rule once The required solution is obtained as the union of the sets P_0 P_1 P_2 etc Let us write P_1 as $r(P_0)$ P_2 as $r(P_1) = r^2(P_0)$ etc

The desired answer is

$$r^*(P_0) = P_0 + r(P_0) + r^2(P_0) + \cdots$$

Now suppose that we define a new recursive rule that is equivalent to k applications of the original recursive rule. If this rule were applied only to the tuple in P_0 we would generate tuples in P_k, P_{2k} and so on but not those corresponding to the generations in between. If instead we were to apply it to the union of P_0, P_1, P_2, etc. up to P_{k-1} then the correct results would be produced at all generations.

$$r^*(P_0) = P_0{}^k + r^k(P_0{}^k) + r^{2k}(P_0{}^k) + \cdots$$

where $P_i{}^j = P_i + P_{i+1} + P_{i+2} + \cdots + P_{i+j-1}$

In addition suppose that we have separately computed P_1, P_2, etc. up to P_{f-k-1}. Then we can use the last k of these as the basis for the k-step recursion and simply add the first f to the result. In other words

$$r^*(P_0) = P_0 + P_1 + \cdots + P_{f-1} + P_f{}^k + r^k(P_f{}^k) + r^{2k}(P_f{}^k) + \cdots$$

4 From Linear Recursion to Transitive Closure

$S^j(X)$ denotes the argument tuple of the antecedent recursive literal in the recursive rule obtained by formally iterating the original recursive rule j times top down. The tuples $S^j(X)$ for different values of j are all of a fixed length. Since there are only finitely many isomorphism classes of argument tuples of a fixed length but an infinite number of values for j it must be the case that there are several $S^j(X)$ that are isomorphic to one another. In particular, if the tuples $S^j(X)$ and $S^{j-k}(X)$ are isomorphic for some positive integers j and k then the tuples $S^{j+mk}(X)$ are all isomorphic for $m = 0, 1, 2, \ldots$. The basic idea is that S^{j+k} is obtained from S^k by k applications of the recursive rule. If k applications of the rule retains the isomorphism then one should expect another k application to retain it as well, and by induction establish the result for any mk.

This idea is formally demonstrated below.

The point to notice at the outset is that if we can find a j and a k such that S^{j-mk} are all isomorphic then a new recursive rule defined as k applications of the original recursive rule using $S^j(X)$ as the argument of the consequent, can be written in the canonical form for transitive closure.

In sections 4.1 and 4.2 we shall see that such a j and k can be determined from the substitution graph. In fact we can choose j and k both equal to the diameter of the substitution graph which we define below.

4.1 The Diameter of the Substitution Graph

Consider the substitution graph for a recursive rule. Since there are no repeated arguments in the consequent each node in the substitution graph can have at most one predecessor. From this, we can deduce (see Lemmas 1-3 in the appendix) that the substitution graph consists only of trees and disjoint directed cycles with trees hanging off the cycles.

Each application of the recursive rule moves' the arguments at the source of each arc in the substitution graph to the destination position of that arc. New arguments are introduced into the positions that have no incoming arc. If the substitution graph for a recursion is a tree after a number of iterations equal to the depth of the tree, say d, the distinguished arguments will all have disappeared (and been supplanted by new arguments that were introduced through the root of the tree). Further applications of the recursive rule will introduce newer arguments still and will not reintroduce the disappeared distinguished arguments. If the substitution graph is a cycle, then the arguments are simply circulated around the cycle, and after a number of iterations equal to the length of the cycle are back in the original positions again. If there are two cycles of lengths d_1 and d_2 respectively with d being the least common

multiple of d_1 and d_2 then after any multiple of d iterations the distinguished arguments will be in their original positions in both cycles Define the *diameter* of the substitution graph to be the least common multiple of all the cycle lengths in the graph that is not less than the depth of all the trees in the graph The diameter d will then have the property that all distinguished arguments are either in the same argument positions or not present at all after any multiple of d applicatons of the recursive rule Furthermore, all variable positions which have a common k^{th} ancestor, for some positive integer k, contain the same argument after k iterations By definition of the diameter, $k \leq d$ Therefore after d iterations the pattern of repetition of arguments has been established

To establish isomorphism we need to exhibit two properties

1 The pattern of arguments must be the same
2 Any common arguments should appear in the same positions

The first property holds for any pair $S^d(X)$ and $S^{d+k}(X)$ for any positive integer k The second property holds for any pair of tuples d iterations apart Therefore we have given a geometric proof of the following

Theorem 1

Given a linear recursive rule r Let S^j denote the argument tuple of the antecedent of the recursive rule r^j obtained by formally iterating j times the rule r Let d be the diameter of the substitution graph for r For any integer $j \geq d$ all the tuples in the sequence S^{j+kd} $k = 0\ 1$ are isomorphic

The theorem can also be proved algebraically, using the algebraic formula for the d^{th} iterate of the substitution map S

Example [2] *A Large Exmaple*

Consider the recursive rule

$p(X_1 \quad X_{15}) - q(Y_1\ Y_2)$
$p(Y_1\ X_2\ X_2\ X_3\ X_6\ X_2\ X_{10}, X_7\ X_7\ X_{11}\ X_7\ Y_2\ X_{11}\ X_{13}\ X_{15})$

where $X = (X_1 \quad X_{14})$ and $Y = (Y_1\ Y_2)$ The substitution graph is shown below

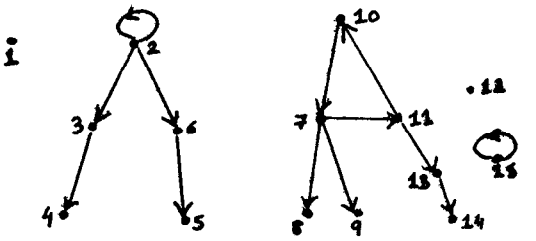

Figure 2. A Large Subsitution Graph

It is evident from a ursory glance at the substitution graph that the largest cycle is of size 3 and the length of the largest chain is just 2 therefore the diameter of this graph is 3 To verify this conclusion obtained graphically let us write the argument tuples $S(X)$ $S^2(X)$ etc obtained by iterating the given recursive rule The first six iterates can be written

$(Y_{1\,1}\ X_2\ X_2\ X_3\ Y_6\ X_2\ X_{10}\ X_7\ X_7\ X_{11}\ X_7\ Y_{1,2}\ X_{11}\ X_{13}\ X_{15})$

$(Y_{2\,1}\ X_2\ X_2\ X_2\ X_2\ X_2\ X_{11}\ X_{10}\ X_{10}\ X_7\ X_{10}\ Y_{2,2}\ X_7\ Y_{11}\ X_{15})$

$(Y_{3\,1}\ X_2\ X_2\ X_2\ X_2\ X_2\ X_7\ X_{11}\ X_{11}\ X_{10}\ X_{11}\ Y_{3,2}\ X_{10}\ X_7\ X_{15})$

$(Y_{4\,1}\ X_2\ X_2\ X_2\ X_2\ X_2\ X_{10}\ X_7\ X_7\ X_{11}\ X_7\ Y_{4,2}\ X_{11}\ X_{10}\ Y_{15})$

$(Y_{5\,1}\ X_2\ X_2\ X_2\ X_2\ X_2\ X_{11}\ X_{10}\ Y_{10}\ X_7\ X_{10}\ Y_{5,2}\ X_7\ X_{11}\ X_{15})$

$(Y_{6\,1}\ X_2\ X_2\ X_2\ X_2\ X_2\ X_7\ Y_{11}\ X_{11}\ X_{10}\ X_{11}\ Y_{6,2}\ X_{10}\ X_7\ X_{15})$

$Y_{k\ 1}$ and $Y_{k,2}$ are the new arguments introduced in the k^{th} iteration for Y_1 and Y_2 It is easy to see that arguments propagate along the arcs of the substitution graph in each iterate with new arguments being introduced in the positions corresponding to nodes with no predecessors The third and sixth iterates are indeed isomorphic

4 2 The Standard Form for an Argument Tuple

We have just established that all tuples S^{d+kd} are isomorphic for $k = 0,1,2,3, \ldots$ where d is the diameter of the graph It is often the case that for a particular rule there is a j smaller than d for which all of the tuples in the sequence S^{j+kd} are isomorphic In other words the pattern of

repetition may stabilize before the d^{th} iteration If the substitution graph consists of only cycles and paths, and all of the arguments in positions on the path nodes are distinct variables the number of iterations required to stabilize the pattern is 0 Hence we define an argument tuple X to be *standard* if X is isomorphic to $S^d(X)$ The standardness of a tuple may be checked graphically from the substitution graph or algebraically from the formula for S

Example [3] *A Large Example Again*
Consider once agin the recursive rule ot example 2 It has a substitution graph ot diameter 3 However a persual ot the iterates that we derived in thatexample shows that the second and fifth iterates are already isomorphic In other words the argument tuple is in standard form after two iterations There is no need to wait until d whoich is 3

It is clear that a d-step rule obtained in canonical form for transitive closure form can be used to evaluate the recursive query Let us understand what the initial j steps required to obtain a standard form tuple require The formal iteration that we are performing in this section is top-down' from the goal In an implementation in which the base relation is given and the computation is performed bottom-up, these j steps must simply be performed at the end

One further observation makes possible some savings in the amount of computation required Notice that one step of the d-step iteration is the same as one of these j iterations except that the rule may have been simplified somewhat by projecting out repeated arguments Therefore we cannot possible lose any answers is we apply the full rule (as in the j iterations) instead of the simplified rule (used in the d-step closure) We make use of this fact to apply merely the j-step rule once to the final closure obtained instead of applying the original rule j times The only place where the original rule must be applied j times is to the original value of the recursive relation

A different way of understanding this issue is to realize that all possible tuples can be derived from the infinite set of n-step rules applied to the initial value of the recursive relation for $n = 0\ 1\ 2$ We compute the first j of these explicitly The rest of them we can write as the application of a j – step rule to an infinite set of rules once more This infinite set of (non-recursive) rules is evaluated as a transitive closure and then the j-step rule applied

5 Repeated Arguments in the Consequent

Denote the argument tuple of the consequent recursive literal by $X = (v_1\ v_2\ \ldots\ v_n)$, denote the argument tuple of the antecedent recursive literal by $Y = (v_1\ v_2\ \ldots\ v_n)$ Suppose that some of the arguments of the consequent are repeated Then the first generation tuples will have the pattern of repetition of the consequent tuple This pattern of repetition may force the second generation tuples to have even more repetitions than the first generation Eventually the pattern of repetitions will stabilize in the sense that a pattern of repetition exhibited by tuples of a generation will also be exhibited by tuples of the next generation Suppose that the stable pattern of repetition is first exhibited by tuples of generation f for some recursive rule In the light of the discussion in section 3 3 we can first compute the generations P_1 through P_{f-1} keep them away and then apply the recursive rule only to P_f with repeated arguments projected out The union of P_0 through P_{f-1} can then be taken with the result of the closure to obtain he final result

The final pattern of repetition and the number may be predicted from a directed graph as given below (This graph can be thought of as the substitution graph with its arcs reversed Since the substitution graph, as we have it in the previous sections, is not well-defined in the presence of repeated arguments in the consequent we choose to define a new graph)

1 Draw a node for each argument position of the recursive literal and label it with the number of the position
2 Draw an arrow from node i to node j if the argument in position i of the recursive antecedent is the same as the argument in position j of the consequent
3 Collapse any nodes without predecessors if the corresponding positions in the consequent contain the same argument Also collapse nodes having the same predecessor Label resulting node with the subset of labels of the original nodes
4 Repeat the following until there are no changes to the graph Collapse nodes having the same predecessor

Then f is one more than the number of iterations needed in step 4 and the positions corresponding to a node in the final graph will contain the same argument in the tuples of P_f P_{f-1} Define the *fan-in number* to be this number f Furthermore the arcs of the graph obtained at the end of this procedure can simply be reversed to obtain exactly the substitution graph for the derived rule with no repeated arguments in the consequent

Example [4]

$p(X X Y Z) - p(Y Z U V) q(U V)$

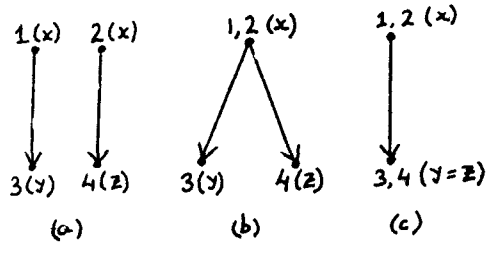

Figure 3 Handling Repeated Arguments in the Consequent

Here f is 2, as can be checked either from the figure or by formally iterating the rule Notice that all tuples of the first generation have their first two arguments equal Therefore when this recursive rule is applied to obtain second generation tuples it could as well be written

$p(X X Y Y) - p(Y Y U V) q(U V)$

But now all tuples in the second generation have their last two arguments equal as well Therefore for the next generation the rule could be written as

$p(X X Y Y) - p(Y Y U U) q(U U)$

Notice that no new arguments were forced equal this time Therefore this same rule applies for all subsequent generations also For reasons of efficiency we may choose to project out repeated arguments and work with a smaller rule

$p(X Y) - p(Y U) q(U)$

It is interesting to note that repeated arguments in the consequent are handled by considering bottom up iteration while repeated arguments in the antecedent are handled by considering top down iteration In fact by an argument similar to the one advanced in section 4 we can claim that we are guaranteed that f can be no more than d the diameter of the substitution graph of the rule

6 Implementation

6 1 The Complete Procedure

Given a linear recursive rule the following procedure can be followed to evaluate it as a transitive closure Let P_0 be the given initial value for the recursive relation (Such an initial value could have been obtained from an exit rule or been otherwise specified)

1 If there are repeated arguments in the consequent of the given recursive rule compute the generations P_0 P_{f-1} by iterating bottom-up f times where f is the fan-in number of the recursive rule and save them The higher generation tuples are the tuples of the recursive relation defined by the original rule with an initial relation P_f Eliminate repeated arguments from this recursive rule Call the new consequent X and the new initial relation

P'_0 The resulting rule r has distinct arguments in the consequent

2 Compute the first $g-1$ generations of the relation defined by r with consequent X and initial value P_0 by bottom up iteration, and save them, where g is the smallest positive integer such that $S^g(X)$ is standard Recall that g will be at most d the diameter of the substitution graph of the rule which has distinct arguments in the consequent

3 Compute the recursive relation Q defined by r with consequent $S^g(X)$ and initial value P'_0 using the d-step rule The d^{th} iterate of r with consequent $S_g(X)$ will be in canonical form for transitive closure and after elimination of repeated variables will be in the basic form for transitive closure Thus the linear recursion in the d-step rule can be computed as a transitive closure

4 Reconstruct Q by replicating the arguments that were projected out in step 2

5 The g^{th} and higher generations of the relation of step 2 can be computed from Q nonrecursively by a single application of the g^{th} iterate of the recursive rule Now (between step 2 and step 5) all of the relation $r(X)$ of step 2 has been computed

6 If the original rule had repeated arguments in the consequent replicate the arguments that were projected out in step 1 and take a union with the tuples already found and saved in step 1 to obtain the final answer

The significant point to note is that d / g are all constants that can be determined from the recursive rule specified independent of the underlying data Therefore, the number of applications of the recursive rule in the above procedure except in step 3 is a constant that can be predicted at the time the rule is written rather than when it is computed Therefore the only real recursion (or the only evaluation of a non-trivial fix-point operation), is in step 3 of the procedure and is no more than the computation of a transitive closure

6 2 Issues of Computational Efficiency

Thus far in this paper we have chosen to write the closure in the most elegant rather than the most computationally efficient way For example we have assumed that there is a single non-recursive predicate obtained as the cartesian product (and/or) join of all the non-recursive predicates involved While such a view is useful from a conceptual perspective, a literal implementation would be computationally inefficient In this section we consider this and such other issues and hope to convince the reader that all the transformations performed on the original recursive expression do not result in a large difficult-to-evaluate transitive closure

Single versus Multiple Non-Recursive Literals

Wherever we have suggested obtaining the cartesian product of the multiple non-recursive predicates first and taking the closure later an actual implementation would likely prefer to reverse the order taking the transitive closure first and computing the cartesian product afterwards Such an evaluation order may be adopted provided that the generation is also recorded along with each tuple In the final step rather than take a cartesian product of these different transitive closures one should compute the multiple-way join based on the generation In other words take the cartesian product only of tuples belonging to the same generation This procedure is reminiscent of the counting" techniques described in [6]

If the different non-recursive literals share variables so that their amalgamation into a single literal would involve a join rather than a cartesian product then the technique just suggested will not work However, it is a well known fact that the join of two or more

relations is usually smaller than the original relations themselves Therefore we may actually prefer taking the join first and computing the closure afterwards in any case

Arguments Common to the Composition Fields

Suppose that there is some v_j that appears in the same position in the recursive literal both in the head and in the body of the rule and also is a part of the non-recursive predicate We can notice that the value of v_j does not change as the recursive rule is applied

We can exploit this to our advantage in the implementation if we split the relation q into several relations, one for each value that v_j can take The transitive closure can be found for each of these relations and then the union can be taken afterwards Using this technique one may be able to evaluate several small closures rather than a single large one Since the effort to compute a closure grows more than linearly with the size of the initial relation this partitioning would result in considerable saving of effort

Decomposition of the Cyclic Component

We have seen that the standard form of the linear recursion consists of a cyclic component and an acyclic component The cyclic component consists simply of a set of arguments that get permuted with these pemutations repeating every d cycles Therefore the transitive closure of the cyclic component is particularly easy to evaluate Therefore the "real" transitive closure that has to be evaluated is only the acyclic (straight line) part of the recursion If one remembers to associate the generation with every tuple in this transitive closure, than one can compute the total closure by taking a join with pattern in the cyclic permutation that would correspond to that generation Thereby, one may get away with a much smaller closure to evaluate

7 Applications

The results presented in this paper have significant theoretical and practical ramifications In this section we outline some applications of our results and also suggest some future research topics

7 1 Analysis

Since every linear recursive expression can be reduced to the basic form for transitive closure analytic techniques developed to study the latter will suffice for the large part For example, moving selections into recursive queries has for long been a topic of interest since the computational savings that can be so derived are considerable However determining when such selections can be moved in has not been easy and several papers (*cf* [9 20]) have dealt with this topic Given the framework presented above such determination becomes much easier In fact we claim that it is possible to completely understand how far the selections in the consequent can be propagated We may assume that the consequent has no repeated arguments after perhaps computing the first $f-1$ generations and then projecting out repeated arguments The selections in argument positions corresponding to nodes on cycles of the substitution graph will persist in the antecedent of all formal iterations of the recursive rule in a predictable and eventually cyclic way The selections in the other argument positions will vanish from the recursive antecedent after at most d iterations Thus if the substitution graph is acyclic an entire recursive relation will have to be computed

Similarly the detection of boundedness can be simplified For if the substitution graph consists only of components containing a cycle, the recursion is bounded In fact at most $2d$ bottom-up iterations are required, and if the original tuple is standard only d iterations are required

7.2 Evaluation

Since every linear recursion has the evaluation of a transitive closure at its core efficient evaluation techniques for transitive closure become indispensable. In fact one need only attempt to optimize the evaluation of transitive closure rather than worry about all possible kinds of linear recursive rules. We feel that the recent efforts to develop efficient techniques for computing transitive closure [10-12, 21] are steps in the right direction. One associated issue that has thus far defied solution is the problem of decomposing a given recursive expression into smaller expressions that would be easier to evaluate. This problem is very hard in general but is a subject of continuing research and we hope that by restricting ourselves to transitive closure meaningful results can be obtained.

7.3 Description

A major impediment in the study of recursive (or fixed-point) queries to databases has been the absence of a convenient way of expressing these queries. One has to state an initial relation, a recursive rule, and the fact that one is interested in recursive application of the rule. The results proved here motivate the inclusion of transitive closure as an extension to ordinary relational algebra. Such an extension is elegantly made in the recently developed α-extended relational algebra [18]. We have shown that every linear recursion can be expressed as a transitive closure and some additional operations already available in relational algebra. Consequently α-extended relational algebra is at least as powerful as linear recursion.

8 APPENDIX: Substitution Graphs and Their Diameter

The crucial characteristic properties of substitution graphs are summarized and proved here. Since substitution graphs exhibit all of the patterns of recursion one can easily construct interesting examples of linear recursive queries by first drawing an interesting substitution graph.

Lemma 1 A directed graph with nodes labeled 1 n for some integer n is a substitution graph for some linear recursive query if and only if every node has at most one immediate predecessor.

Proof In a linear recursive rule which contains distinct distinguished variables in distinct positions each distinguished variable which appears in the recursive predicate comes from exactly one position in the goal predicate. Thus each node in the substitution graph determined by a linear recursive relation has at most one predecessor. Conversely given a directed graph with nodes labeled by 1 n in which each node has at most one predecessor one can construct a linear recursive rule with that graph as its substitution graph.

In fact every node in a substitution graph has at most one k^{th} predecessor for every positive integer k. For by a k^{th} predecessor we mean a node reached by going backwards along k edges.

This characterization determines a convenient restriction on the structure of the connected components of the underlying undirected graph. Since we are considering connectedness properties of the underlying underected graph one might initially think that such a graph might contain a (undirected) cycle. Such doubts are dispelled by the following.

Lemma 2 A connected component of a substitution graph contains at most one (undirected) cycle. Any undirected cycle is in fact a directed cycle.

Proof Suppose the component contains an undirected cycle, if the cycle were not a directed cycle, some node on the cycle would have 2 edges entering it which is impossible by the characterization of substitution graphs. Now suppose a component contained 2 directed

simple cycles If both cycles contain node j they must both contain a predecessor of node j Since there is only one such predecessor both cycles contain the unique predecessor of node j Continuing this argument we see both cycles contain each other If the 2 cycles do not intersect, there is an undirected path from one to the other since they are contained in the same component We may assume the path contains exactly one node from each cycle say n_1 and n_2 respectively The edges on the path must be directed away from the cycles so that neither n_1 nor n_2 has more than one entering edge Hence the path contains some node with 2 entering edges which again contradicts the fact that each node has at most one predecessor

Naughton in [7] stated a result equivalent to Lemma 2 for his argument/variable graphs

Lemma 3 A component of a substitution graph is either a rooted tree or the union of a simple directed cycle with 0 or more trees rooted at nodes of the cycle, but otherwise disjoint from the cycle

Proof If a component is acyclic in the undirected sense we must show there is a unique node with no predecessors If there were 2 such nodes there would be an undirected path from one node to the other node The end edges would be directed away from the end nodes Hence some node on the path would have 2 immediate predecessors a contradiction If the component contains a cycle then deleting the edges in the cycle must leave an acyclic graph by Lemma 2 which as we have just shown is a union of rooted trees The roots must be on the cycle since the original component was connected

Lemma 3 implies that there are two natural subclasses of substitution graphs to consider namely those with only acyclic components and those consisting of components which contain a cycle

Remark If no distinguished variable is repeated in the recursive predicate then no node has more than one successor so the components are either a single node a non-trivial path or a simple cycle

REFERENCES

[1] H Gallaire J Minker and J Nicolas Logic and Databases A Deductive Approach *ACM Computing Surveys 16* 2 (June 1984) 153-185

[2] F Bancilhon and R Ramakrishnan An Amateur's Introduction to Recursive Query Processing Strategies *Proc ACM-SIGMOD 1986 Int'l Conf on Management of Data*, Washington D C May 1986 16-52

[3] L Henschen and S Naqvi, On Compiling Queries in Recursive First Order Databases *J ACM 31* 4 (Jan 1984) 47-85

[4] F Bancilhon Naive Evaluation of Recursively Defined Relations Tech Rept DB-004-85 MCC Austin Texas 1985

[5] Y E Ioannidis A Time Bound on the Materialization of Some Recursively Defined Views *Proc 11th Int'l Conf Very Large Data Bases* Stockholm Sweden Aug 1985 219-226

[6] F Bancilhon D Maier Y Sagiv and J Ullman Magic Sets and Other Strange Ways to Implement Logic Programs *Proc 5th Symp Principles of Database Systems* Cambridge Massachusetts March 1986 1-15

[7] J Naughton Data Independent Recursion in Deductive Databases, *Proc 5th Symp Principles of Database Systems*, Cambridge Massachusetts, March 1986, 267-279

[8] Y E Ioannidis and E Wong, An Algebraic Approach to Recursive

Inference *Proc 1st Int'l Conf Expert Database Systems* Charleston South Carolina April 1986 209-224

[9] P Devanbu and R Agrawal Moving Selections into a Class of Least Fixpoint Queries AT&T Bell Laboratories Technical Memorandum 1986

[10] P Valduriez and H Boral Evaluation of Recursive Queries Using Join Indices *Proc 1st Int l Conf Expert Database Systems* Charleston South Carolina April 1986 197-208

[11] Y E Ioannidis On the Computation of the Transitive Closure of Relational Operators *Proc 12th Int'l Conf Very Large Data Bases* Kyoto Japan Aug 1986 403-411

[12] R Agrawal and H V Jagadish Direct Algorithms for Computing the Transitive Closure of Database Relations submitted to *VLDB* 1987

[13] S Warshall A Theorem on Boolean Matrices *J ACM 9* 1 (Jan 1962) 11-12

[14] L E Thorelli An Algorithm for Computing All Paths in a Graph *BIT 6* (1966) 347-349

[15] P Purdom A Transitive Closure Algorithm *BIT 10* (1970) 76-94

[16] H S Warren A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations *Commun ACM 18* 4 (April 1975) 218-220

[17] C P Schnorr An Algorithm for Transitive Closure with Linear Expected Time *SIAM J Computing 7* 2 (May 1978) 127-133

[18] R Agrawal Alpha An Extension of Relational Algebra to Express a Class of Recursive Queries. *Proc IEEE 3rd Int'l Conf Data Engineering*. Los Angeles California Feb 1987

[19] E F Codd A Relational Model of Data for Large Shared Data Banks *Commun ACM 13* 6 (June 1970) 377-387

[20] M Kifer and E L Lozinskii Filtering Data Flow in Deductive Databases *Int'l Conf Database Theory* Rome Italy Sept 1986

[21] H Lu K Mikkilineni and J P Richardson Design and Evaluation of Algorithms to Compute the Transitive Closure of a Database Relation *Proc IEEE 3rd Int'l Conf Data Engineering* Los Angeles California Feb 1987 112-119