A TIME BOUND ON THE MATERIALIZATION OF SOME RECURSIVELY DEFINED VIEWS

Yannis E. Ioannidis

Department of Electrical Engineering and Computer Science
University of California
Berkeley, CA 94720

Abstract

A virtual relation (or view) can be defined with a recursive statement that is a function of one or more base relations. In general, the number of times such a statement must be applied in order to retrieve all the tuples in the virtual relation depends on the contents of the base relations involved in the definition. However, there exist statements for which there is an upper bound on the number of applications necessary to form the virtual relation, independent of the contents of the base relations. Considering a restricted class of recursive statements, we give necessary and sufficient conditions for statements in the class to have this bound.

1. INTRODUCTION

In the past few years major attempts have been made to improve the power of database systems, in particular those based on the relational model (see [Codd70]). A significant part of this effort has been in the direction of the formalization, design and development of deductive databases. As defined in [Gall84] , "a deductive database is a database in which new facts may be derived from facts that were explicitly introduced". A very important difference between a deductive and a conventional relational database is that in the former new facts may be derived recursively. This very characteristic of deductive databases is what makes query processing a difficult task in such an environment. The main problem that arises is how to detect the point at which further processing will give no more answers to a given query. Many researchers have studied and proposed solutions to this termination problem for various cases (see, for example, [Naqv84] , [Reit78] and [Chan81]). However no single solution is known for the general problem.

This research was supported by the National Science Foundation under Grant ECS-8300463

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

A common characteristic among all the proposed solutions that we are aware of is that the termination condition relies on the data explicitly stored in the database. In general this is necessary. However, there are some cases where a termination condition exists, which is independent of the particular instance of the database. The purpose of this paper is to identify and characterize these cases. Restricting ourselves to a particular class of recursive statements, we give necessary and sufficient conditions for the existence of a data-independent termination condition.

We assume that the reader has some familiarity with mathematical logic and graph theory, although nothing extremely involved from these fields will be needed. Nevertheless, we are going to use some of their notions without definition. The first few chapters of any standard text in mathematical logic (e.g. [Ende72]) and graph theory (e.g. [Bond76]) provide the necessary background. Furthermore we assume that the reader is familiar with relational databases at the level of [Date82] . Finally, we would refer the reader to [Gall78] and [Gall81] as extremely valuable sources of information on the relationship between mathematical logic and deductive databases.

The paper is organized as follows. In Section 2 we give the formal framework of a deductive database that we will be considering. Our investigation is restricted to a subset of all possible deductive databases. We outline all the restrictions we are imposing on the database and explain the reasons for doing so. In Section 3 we introduce some examples of cases where even though data is derived recursively, the termination condition is known *a-priori* (i.e. it does not depend on the explicitly stored data). Section 4 contains the description of the graph model we used as a tool to derive our results. In Section 5, we state the main result of this paper: the necessary and sufficient conditions for a termination condition to exist that is independent of the data explicitly stored in the database. Furthermore we illustrate our result with a number of characteristic examples. The theorem is presented here without any formal proof. For a detailed analysis and proof we refer the reader to [Ioan85] . In Section 6 we discuss the importance of our results and investigate ways in which they can be used to speed up query processing in

deductive databases. Finally, in section 7 we summarize our results and discuss more problems for future work in the area.

2. ASSUMPTIONS

The following definitions about first-order formulas ([Ende72]) will be useful in our analysis.

Definition 2.1: A first-order formula is equivalent to a *Horn clause* if and only if it is of the form

$$A_1 \wedge A_2 \wedge \cdots \wedge A_n \rightarrow C$$

with all the variables appearing in the formula being (implicitly) universally quantified.

The formula to the left of $\rightarrow$ will be called the *antecedent* and that to the right of $\rightarrow$ the *consequent*. Each one of $C, A_1, A_2, \ldots, A_n$ is an atomic formula (see [Ende72]), i.e. it is of the form

$$P(t_1, t_2, \ldots, t_n)$$

where P is a predicate symbol and t_i, $1 \le i \le n$, is a term (a variable symbol or a constant symbol or a function symbol "applied" on one or more terms). Finally, a Horn clause is recursive when the predicate that appears in the consequent appears at least once in the antecedent as well. Throughout the paper we will be using the terms "formula" and "statement" indistinguishably. We will also alternate between the terms "predicate" and "relation", in light of the discussions in [Gall78] .

Definition 2.2: Two variables x, y *appear under the same predicate* in a statement if and only if there is an atomic formula $P(\ldots,x,\ldots,y,\ldots)$ appearing in the statement, where P is a predicate symbol.

Definition 2.3: Consider a recursive statement which is equivalent to a Horn clause. The sole predicate appearing in the consequent of the statement will be called the *recursive predicate* of the statement. Any other predicate in the statement will be called *non-recursive*.

Definition 2.4: A variable will be called *consequent* if and only if it appears under the recursive predicate in the consequent of the statement. Otherwise it will be called *antecedent*.

We consider a deductive database to be a relational database (in the sense of [Codd70]) enhanced with a set of Horn clauses. If there is some recursive statement or a set of mutually recursive statements appearing in the database, then the termination problem mentioned in Section 1 arises. We will examine this problem with respect to the processing of a single recursive statement only.

We restrict our attention to recursive statements that satisfy the following conditions:

1) The recursive predicate of the statement appears only once in the antecedent.
2) There are no function symbols in the statement.
3) There are no constant symbols in the statement.
4) No variable appears more than once under the recursive predicate in the consequent. Furthermore, no subsequence of the variables appearing under the recursive predicate in the consequent is a permutation of the corresponding subsequence of the variables in the recursive predicate in the antecedent.

Our motivation behind restriction (1) is simplicity. Having more than one appearance of the recursive predicate in the antecedent severely complicates our analysis. Since most of the recursive statements expected in a real world system will have the recursive predicate appearing only once in the antecedent, we believe that assumption (1) is reasonable. Function symbols appearing in a recursive statement may lead to infinite relations. For example, consider the following recursive statement containing the '+' function:

$$P(x) \rightarrow P(x+1)$$

Suppose that initially P contained the single tuple $<1>$. It is clear that the above statement makes P an infinite relation containing all the positive integers. Situations like that are not easily handled in a database environment, if at all; to avoid them we have imposed restriction (2). The last two restrictions were imposed for the sole purpose of getting a uniform result. We speculate that it will not be very difficult to remove them, thereby generalizing our results. In fact, considering a recursive statement that does contain constant symbols, we may remove them by performing selections and projections on the relations involved (see [Codd70]). The new statement is free of constant symbols and if applied to the new set of relations produced by the operations mentioned above, will give the same result as if the original statement was applied on the original relations. Regarding restriction (4), it may appear somewhat artificial, but its meaning will become clear shortly, when we will describe the way we model a recursive statement.

A final point worth mentioning here is that without loss of generality we may assume that there are no equalities in the statement. If there is any equality between two variables, we may easily remove it by replacing one of its variables with the other wherever it appears in the statement. It is clear that the new statement is equivalent to the initial one.

Definition 2.4: A recursive statement will be called *simple* if and only if it satisfies conditions (1) through (4) above and does not contain any equality symbol.

3. SOME EXAMPLES

Consider the following simple recursive statement α:

$$\alpha: \quad P(x) \wedge Q(x,y) \rightarrow P(y)$$

Relation Q is a base relation in the system (that is, it is stored explicitly), whereas P is a derived relation. It is clear that in addition to α above, there has to be some non-recursive way to get some initial tuples into P. As an example assume that this is done with β:

$$\beta: \quad R(x) \rightarrow P(x)$$

Assume that R is a base relation as well. A natural way of thinking about the processing of α is iteration. In particular, the statement is applied once on the initial contents of the relations involved and produces some new tuples for P. This process is repeated for these new tuples and then again, until no new tuples are produced. It is obvious that in general there is no upper bound on the number of times this process has to be repeated in order to get all the derivable tuples for P. If we consider Q to represent a directed graph and R to contain some nodes of the graph, then P comes to contain all nodes reachable from those in R. At step i of the iterative process described above, we insert into P all those nodes of the graph that are reachable from some node in R through a path of length i. Since the graph may contain arbitrarily long paths, it is not possible to know in advance how many iterations will be needed.

As another example of a simple recursive statement, consider γ:

$$\gamma: \quad P(z) \wedge Q(z) \wedge R(y) \rightarrow P(y)$$

where Q and R are base relations. Clearly, one application of the statement is enough, regardless of the initial contents of the relations P, Q and R. Statement γ derives for P all the tuples in R, as long as there is initially one tuple in P that joins with (that is, is equal to) some tuple in Q. Any further step in the iteration will fail to produce any new tuples for P. So for γ, unlike α, there exists an upper bound on the number of times the statement has to be applied to derive all the tuples possible in the recursive relation, that number being equal to 1.

As a third example consider δ:

$$\delta: \quad P(z,x) \wedge Q(y) \rightarrow P(x,y)$$

with Q being a base relation. In this case we are taking the cartesian product of the projection on the second attribute of the initial copy of P with Q. However one step is not enough for δ. One more step will be needed, where actually the cartesian product of Q with itself will be derived for P. Nevertheless, there will be no need for a third step. Further processing will only continue producing the cartesian product of Q with itself. Therefore δ, like γ, has an upper bound on the number of times it needs to be applied, only that now the tight upper bound is equal to two. This is not to say that the second step of the iteration will always produce new tuples for P. In fact, if Q is initially empty, not even the first step will be needed. However the point is that there exists an instance of Q and P that will need two steps, whereas there exists no instance of these relations that will need three.

The examples given above indicate that the way in which the variables appearing in the statement are connected with each other through the predicates, plays an important role on whether an upper bound on the number of iterative steps needed to produce all derivable tuples exists or not. In order to study the properties of these statements we developed a graph model for them, which reflects this connection among the variables. The description of this model is the subject of the next section.

4. THE MODEL

Suppose that we are given a simple recursive statement. We will model this statement by a labeled, weighted, directed graph constructed as follows:

(i) To every variable appearing in the statement we associate a node in the graph.

(ii) For every pair of variables x,y that appear under the same non-recursive predicate Q in the statement there is a labeled undirected edge $(x - y)$ in the graph between the corresponding two nodes x,y, for each such predicate Q. The label of the edge is Q and its weight is 0.

(iii) For every pair of variables x,y such that x appears under the recursive predicate P in the antecedent and y appears in the corresponding position of the recursive predicate in the consequent, there is a directed edge $(x \rightarrow y)$ in the graph from node x to node y with weight 1 and its inverse edge $(y \rightarrow x)$ with weight -1. Each directed edge has label P.

The graph constructed in this way from a simple recursive statement α will be called the ***α-graph***. The subgraph induced on the α-graph by the undirected edges defined in (ii) will be called the ***static α-graph***. The spanning subgraph of the α-graph with edge set its directed edges defined in (iii) will be called the ***dynamic α-graph***. Finally, the length of a path (cycle) in the graph is defined to be the sum of the weights of the edges along the path (cycle). Regarding undirected edges, they can be traversed in both directions, as if there were two opposite directed edges.

As an example consider the following simple recursive statement:

$$\alpha: \quad P(z,w) \wedge Q(z,x) \wedge R(w,u) \wedge S(u,x,y) \rightarrow P(x,y)$$

The α-graph is shown in figure 4.1.

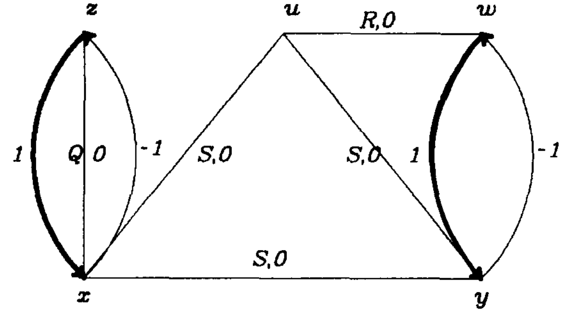

Fig. 4.1 : The α-graph

We can now see the meaning of restriction (4) in Section 2. All it says is that the dynamic subgraph of a simple recursive statement (restricted on the positive edges) is a forest. This has the implication that there is at most one path from any node to any other node in the subgraph, which proved to be crucial for the accuracy of our results.

For those familiar with the unification algorithm (see [Robi65]), we would like to indicate an important relationship between that and the graph model analyzed above. Unification is a first-order theorem proving algorithm. The iterative process used for recursive statements in Section 2, is equivalent, with respect to the final outcome, to a unification process. In particular, consider two copies of the statement, with distinct variable symbols for all the antecedent variables. That is, consider α as given above and α' as given below:

$$\alpha': \quad P(z',w') \wedge Q(z',x) \wedge R(w',u') \wedge S(u',x,y) \rightarrow P(x,y)$$

Clearly α is equivalent to α', since all we did was to change some variable names. We can now unify the recursive predicate in the antecedent of the first copy with that in the consequent of the second copy (the unification algorithm would work with the statements put in clause form, but its actions are equivalent to the ones we describe here). The resolvent is a new simple recursive statement, which if applied on the initial instance of the recursive predicate, will give exactly the same result with the application of the original statement on the outcome of the first step of the iterative process. For our example the resolvent comes out to be:

$$P(z',w') \wedge Q(z',z) \wedge R(w',u') \wedge S(u',z,w) \wedge Q(z,x) \wedge R(w,u) \wedge S(u,x,y) \rightarrow P(x,y)$$

Regarding unification, the dynamic subgraph of a simple recursive statement captures something very important. Namely, it shows the substitution of the

variables that one has to make to unify the two literals in the two copies of the statement. For every positive directed edge of the graph, the tail should be substituted in the second copy for the head, to obtain the resolvent. In the example above, z was substituted for x and w was substituted for y, which is exactly what the directed edges $(z \to x)$ and $(w \to y)$ in figure 4.1 indicate. We will not be directly referring to the unification algorithm, but the ideas behind it have a significant impact on our analysis.

Finally, there is a notational comment we would like to make about the graph model described above. According to the definition, there is a one to one correspondence between the positive and the negative directed edges. The positive ones alone are enough to carry all the information captured by the directed edges in the graph. In the remainder of this paper we will be referring to the dynamic subgraph as containing the positive edges of the graph only, the negative ones implicitly assumed only whenever the length of a path is discussed. Likewise, in all the figures we will draw the positive edges only. Finally, since the weight of some edge is easily determined from whether it is undirected (weight zero) or directed (weight one), we will put no weights on the edges.

5. THE PROBLEM

Let the following be a simple recursive statement.

$$P(x_1,x_2,\ldots,x_m) \wedge \beta \to P(y_1,y_2,\ldots,y_m) \qquad (1)$$

The subformula β is a conjunction of atomic formulas, none of which involves the predicate P, and all variables are assumed to be universally quantified. There are two equivalent ways of expressing (1) in a nonrecursive way.

- The above statement may be viewed as equivalent to the following infinite sequence of statements:

$$P_0(x_1,x_2,\ldots,x_m) \wedge \beta \to P_1(y_1,y_2,\ldots,y_m)$$
$$P_1(x_1,x_2,\ldots,x_m) \wedge \beta \to P_2(y_1,y_2,\ldots,y_m)$$
$$P_2(x_1,x_2,\ldots,x_m) \wedge \beta \to P_3(y_1,y_2,\ldots,y_m)$$
$$\ldots\ldots\ldots\ldots$$

In the above statements, P_0 denotes the initial contents of P, P_1 denotes the tuples "inserted" into P after applying the recursive statement once, P_2 denotes the tuples "inserted" in P after applying the recursive statement on the new tuples produced by the previous application, and so on. The final result for relation P is $\bigcup_{i=0}^{\infty} P_i$. The i-th statement above will be called the ***i*-th application** of statement (1). Note that this infinite non-recursive expression of (1) actually reflects the iterative process to materialize P, along the lines of our discussions in the previous sections.

- Statement (1) may be also viewed as equivalent to the statements

$$P_0(x_1,x_2,\ldots,x_m) \wedge \beta_0 \to P_1(y_1,y_2,\ldots,y_m)$$
$$P_0(x_1^{(1)},x_2^{(1)},\ldots,x_m^{(1)}) \wedge \beta_1 \wedge \beta_0 \to P_2(y_1,y_2,\ldots,y_m)$$
$$P_0(x_1^{(2)},x_2^{(2)},\ldots,x_m^{(2)}) \wedge \beta_2 \wedge \beta_1 \wedge \beta_0 \to P_3(y_1,y_2,\ldots,y_m)$$
$$\ldots\ldots\ldots\ldots$$

where, for all $i \geq 0$, it is $\beta_i = \beta[\vartheta_i]$ and $P_0(x_1^{(i-1)},x_2^{(i-1)},\ldots,x_m^{(i-1)}) = P_0(x_1,x_2,\ldots,x_m)[\vartheta_i]$, with ϑ_i some substitution of the variables in β, the details of which will not concern us for the moment. Note that ϑ_0 maps each variable to itself. In a similar but somewhat different way than for the applications of (1), the i-th statement above will be called the ***(i-1)*-th expansion** of statement (1), so that the first of these statements, which is actually the statement itself, is the 0-th expansion. Each one of these expansions is applied on the initial contents of P. Clearly, the P_i's above are the same as the ones in the "application" view of the recursive statement, i.e. the tuples produced by the i-th application of (1) for P_i, are the same as those produced by the $(i-1)$-th expansion of (1) for P_i. In the end P is again equal to $\bigcup_{i=0}^{\infty} P_i$. Note that the antecedent of each expansion is equal to that of the previous expansion with the recursive predicate being replaced by yet another instance of the antecedent of the original statement (with different variables, of course). In terms of the unification algorithm mentioned in Section 4, the k-th expansion of a simple recursive statement α is the resolvent of its $(k-1)$-th expansion and α itself. In the sequel we will be referring to the k-th expansion of a recursive statement α as α_k. Since the statement itself is its own 0-th expansion we have that $\alpha = \alpha_0$.

In both cases above, the initial statement becomes equivalent to an infinite number of nonrecursive statements. However, since in a database environment all the relations are finite, and because of the fact that we are considering simple recursive statements only, which contain no functions, after some point the nonrecursive statements will stop producing any new tuples for P, and therefore the whole process eventually terminates. Moreover the process terminates exactly when some i-th application (or the corresponding $(i-1)$-th expansion of (1)) fails to produce any new tuples for the first time.

This is an appropriate place for the following definitions.

Definition 5.1: Let α be a simple recursive statement. The ***rank*** of α is defined to be the smallest i such that α_i does not produce any tuple not already contained in some P_j for $0 \leq j \leq i$.

Note that, in general, the rank of α is dependent on the contents of the relations involved in α.

Definition 5.2: A simple recursive statement will be called ***bounded*** if and only if there exists a finite upper bound on its rank independent of the contents of the relations involved in the statement.

In view of the previous definitions we can pose our problem as the following question:

When is a simple recursive statement bounded?

We are also interested in finding this upper bound in the cases it exists. The answer to this question is given by the following theorem.

Theorem: Let α be a simple recursive statement. Statement α is bounded if and only if the α-graph contains no cycle of non-zero length. In that case a tight upper bound on the rank of α is equal to the maximum length of any path in the α-graph.

We have mentioned earlier that a detailed analysis and proof of this theorem may be found in [Ioan85] . However, we will attempt to sketch the important steps in our proof there, by means of some examples.

5.1. SUFFICIENCY OF THE CONDITION

Consider formula γ from Section 3:

$$\gamma: \quad P(z) \wedge Q(z) \wedge R(y) \to P(y)$$

Figure 5.1 shows that the γ-graph contains no cycles at all (excluding that formed through the implicitly assumed negative edge, which again is of length zero).

Fig. 5.1 : The γ-graph

Furthermore the maximum length of any path in the graph is 1. In our discussion in Section 3, we have concluded that one step of the iteration process is enough to derive all possible tuples in P, which is in perfect agreement with the theorem above.

The fact that the rank of γ is bound by 1, can be seen from the first expansion of γ, which is

$$\gamma_1 : \quad P(z') \wedge Q(z') \wedge R(z) \wedge Q(z) \wedge R(y) \rightarrow P(y)$$

If we substitute z for z' and z' for z in γ_1, the antecedent of γ_1 becomes equal to the antecedent of γ with some additional atomic formulas conjuncted to it. The antecedent of γ_1 being strictly more restrictive than that of γ, implies that any tuple derived from the former is also derived from the latter. Thus γ_1 need never be considered.

The same conclusion could be drawn, by looking at the γ- and the γ_1-graphs, as they appear in figures 5.1 and 5.2 respectively.

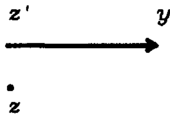

Fig. 5.2 : The γ_1-graph

We can see that the γ-graph is isomorphic to a subgraph of the γ_1-graph. The isomorphism preserves the consequent variables as well as the edges of the dynamic γ-graph. As for the antecedent variables the isomorphism maps them according to the substitution mentioned before, that makes the antecedent of γ part of that of γ_1.

These properties of the expansions and their graphs do not follow from any specific characteristic of γ, but only from the fact that the γ-graph is free of non-zero length cycles. To illustrate this, we will now consider a much more complicated example. Consider the simple recursive formula α:

$$P(u_1,u_2,u_4,u_4,y) \wedge Q(u_1,u_2) \wedge R(u_2,u_3,x) \wedge S(w,z) \rightarrow P(v,w,x,y,z)$$

The α-graph appears in figure 5.3.

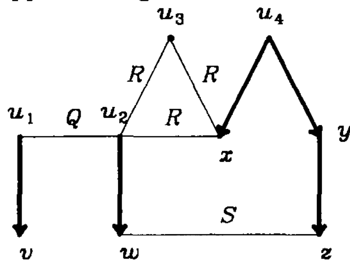

Fig. 5.3 : The α-graph

All the cycles in the α-graph have zero length, including those formed through the negative directed edges, which according to our convention are not drawn (going along a negative edge can be thought of as going along a positive edge in the opposite direction and inversing the weight). Hence, according to our theorem, α is bounded. The maximum length of any path in the graph being 2, we may conclude that α_2 is redundant, i.e. two steps are enough in the iterative process to get the final result for P.

This becomes apparent if we look at the α_1- and the α_2-graphs. They are shown in figures 5.4 and 5.5 respectively.

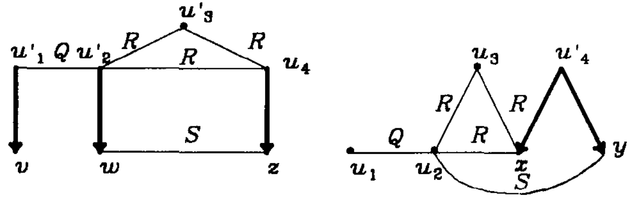

Fig. 5.4 : The α_1-graph

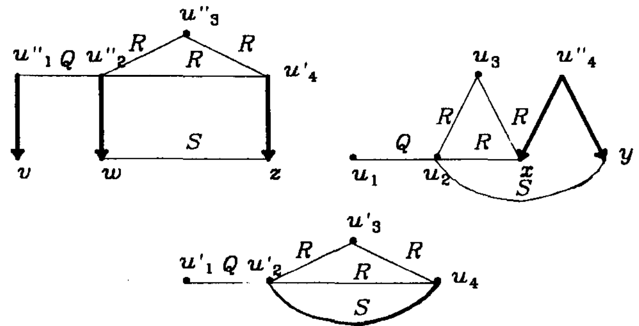

Fig. 5.5 : The α_2-graph

The α_1-graph in figure 5.4 shows two (connected) components, instead of one that initially appeared in the α-graph. Furthermore, the latter is not isomorphic to any subgraph of the former, which implies that there are some instances of the relations involved in α that will make α_1 produce some tuples that are not produced by α. Hence α_1 is necessary.

On the other hand the α_2-graph in figure 5.5 has three components, one more than the α_1-graph. Two of these components are isomorphic to those in the $\alpha_{1\text{-graph}}$. Moreover, the isomorphism has all the desired properties, i.e. it maps consequent variables to themselves and preserves the labeling of the static edges. For α_1 and α_2 this means that changing the antecedent variable names in the latter appropriately, its antecedent becomes strictly more restrictive than that of the former. The rank of α is bounded by 2, therefore α_2 is not necessary.

Notice that the number of components increased in the first example with γ as well. In fact, this is true for all graphs free of non-zero length cycles. If, for example, the graph of the initial statement is connected, then each expansion comes with one more component than the previous one. At some point, we get a component that contains no directed edges, i.e. no consequent variables. This expansion is exactly the first one that is redundant and it determines the bound on the rank of the initial statement.

Another general characteristic of the graphs of bounded statements, which is proved in [Ioan85] , is that the last expansion that is significant (regarding the production of new tuples), is the first one with the maximum length of any path in its graph being 1. This may be seen in both the γ- and the α_1-graphs above.

5.2. NECESSITY OF THE CONDITION

We will now attempt to illustrate with an example, that the condition given in the theorem for bounded statements is as tight as possible. Consider the

statement below:

$$\beta : P(u_1,w,u_2,u_3) \wedge Q(w,u_2) \wedge R(y,u_3) \wedge S(x,z) \rightarrow P(w,x,y,z)$$

The β-graph appears in figure 5.6.

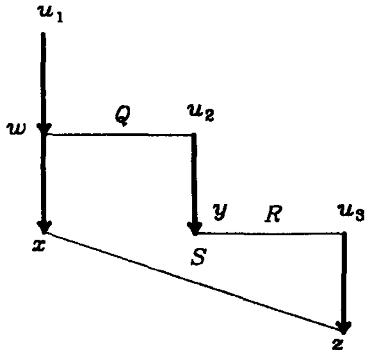

Fig. 5.6 : The β-graph

The β-graph contains a cycle of length 1, namely ($w \rightarrow u_2 \rightarrow y \rightarrow u_3 \rightarrow z \rightarrow x \rightarrow w$). Recall that the edge ($x \rightarrow w$) with weight -1 does exist, even though it is not shown in the figure, and also that an undirected edge can be traversed in both directions. Hence, according to our theorem, β cannot be bounded. The expansions of β become quite complicated and difficult to read, that is why we will attempt to convince ourselves on the unboundedness of β by looking at the graphs of these expansions. The β_1- and β_2-graphs appear in figures 5.7 and 5.8 respectively.

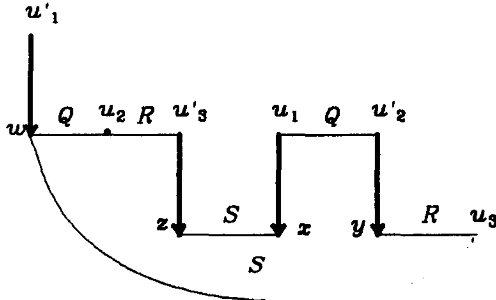

Fig. 5.7 : The β_1-graph

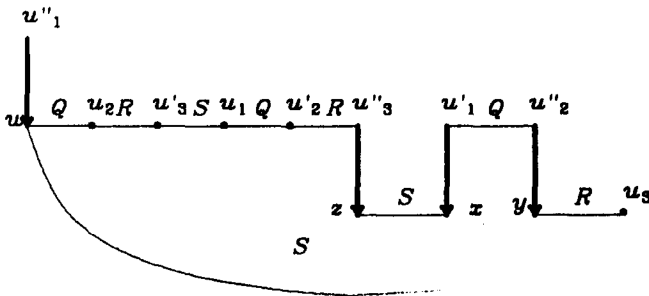

Fig. 5.8 : The β_2-graph

Looking at the way the graph changes as we move from one expansion to the next, we see a distinct difference between what happened in the previous cases and what happens now. Instead of having the number of components increase, we continue to have a single component, but the original cycle becomes larger and larger, in terms of the number of the undirected (static) edges in it. This continues, no matter how many expansions we perform. Clearly, for two cycles to be isomorphic, they have to contain the same number of edges. Since no expansion can have a graph which is isomorphic to a subgraph of any other expansion, there can be no upper bound on the number of them that are significant for the result.

In general, as shown in [Ioan85] , every cycle of length 1 in a graph increases in size from one expansion to the next. Cycles of length greater than 1, have a more complicated behavior. The significant characteristic of it is that at some expansion they break into multiple cycles of length 1. It is not difficult to show that the boundedness property of a statement is inherited to all expansions of it and vice-versa. From this, combined with the general behavior of cycles mentioned above, follows the conclusion that if a statement has a non-zero length cycle in its graph it is not bounded.

The condition of the theorem given in the beginning of the section is sufficient for a statement to be bounded even when restrictions (3) and (4) are removed. However it is not necessary. For example consider the trivial example

$$P(y,x) \rightarrow P(x,y)$$

This statement is not simple. It violates restriction (4) by having a subsequence of the variables under the recursive predicate in the consequent being a permutation of the corresponding variables in the antecedent. The graph of the statement appears in figure 5.9.

Fig. 5.9 : Graph violating restriction (4)

As expected the dynamic graph (restricted to the positive edges), which in this case is equal to the whole graph, is not a forest. Even though it is clear that the statement is bounded with bound 1, the graph contains a cycle of length 2, thus violating our theorem. Future work should attempt to generalize the condition of the theorem to include statements like this as well.

6. APPLICATIONS

Besides its theoretical interest, our result may have considerable implications on how general recursive statements can be processed in a deductive database environment. We have no specific results in that direction but we can speculate that many recursive statements can be decomposed into "smaller" ones, some of which are bounded. Our unbounded statements will be smaller than the initial one and this will result in faster processing. Furthermore, the parts of the result that correspond to bounded statements will be obtained in a bounded number of steps independent of the rest of the statement. This should result in greater efficiency, since the processing of the original statement may involve many more steps than the bound of the bounded statements. Processing the original statement in its initial form would recompute the same things again and again. Of course there will be some overhead in the end

to combine the results of the various substatements in a way that they produce the same result as the original statement. In many cases though, the effort will be worth some net savings in computational cost.

As an example of such a decomposition consider the following statement

$$P(z,w) \wedge Q(z,w) \wedge R(x,y) \wedge S(z,x) \rightarrow P(x,y)$$

The graph of this statement is shown in figure 6.1.

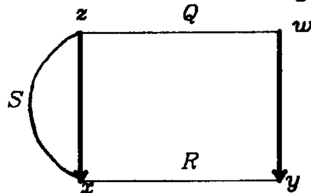

Fig. 6.1 : Graph of decomposable statement

The statement can be decomposed into the two statements

$$P_1(z,w) \wedge Q(z,w) \wedge R(x,y) \rightarrow P_1(x,y)$$

$$P_2(z) \wedge S(z,x) \rightarrow P_2(x)$$

The corresponding graphs of the two statements appear in figure 6.2.

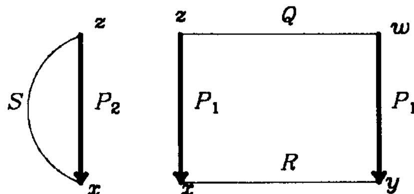

Fig. 6.2 : Graphs of statements after decomposition

As we can see the first statement is bounded, with bound 1, while the second is unbounded. Processing the two statements separately and then combining the two results may affect the processing time significantly.

As for a single statement, the information that it is bounded might prove to be useful as well. Its definition can be expressed non-recursively in a finite form. This makes applicable all the tools used in conventional relational databases to find fast access paths to process the statement. It also makes it much easier to compile such an access path compared to the effort needed for a general unbounded statement (e.g. see [Naqv84]). Finally, when we know that a statement is bounded, with say bound $\bar{n}$, we never need to process the statement for an $(\bar{n}+1)$-th time, only to discover that no new tuples are produced. For statements with small bounds, say 1 or 2, this may prove to be quite significant. We should also note that, even though we have examined the problem of bounded recursive statements in a deductive database context, our results apply to other similar environments too, like those based on the PROLOG programming language.

All the above lead us to the conclusion that the existence of bounded recursive statements and our ability to characterize them is an important step towards processing efficient algorithms for recursion.

7. CONCLUSIONS

We have considered a restricted class of recursive statements in the context of a deductive database. We have demonstrated that some such statements are amenable to an equivalent finite nonrecursive expression, i.e. using the first n expansions of the statement, where n is its bound. By modeling such a statement with a weighted graph, we have shown that the property that the statement can be expressed in a finite way is equivalent to the property that the graph has no cycles of non-zero length. Finally, we have indicated some possible implications of our result in the construction of efficient algorithms to process recursive statements.

There are many things left to be done in the future. We are currently working on obtaining necessary and sufficient conditions for more general classes of recursive statements, by removing some of the restrictions (1) to (4) of section 2. We believe that this should not be difficult for restrictions (3) and (4) and partly for restriction (2). As an even more important task for the future we consider the study of unbounded recursive statements. We are currently investigating the possibility of decomposing such a statement into smaller ones some of which are bounded, in the way it was demonstrated in section 6. Finally, much work needs to be done for unbounded non-decomposable recursive statements.

Acknowledgements: I am deeply indebted to Timos Sellis for the innumerable discussions I had with him, without which this paper may never have existed. I would also like to give thanks to Prof. E. Wong for his valuable help and to Eric Hanson and Brad Rubenstein for their useful comments on earlier drafts of this paper.

8. REFERENCES

[Bond76]
Bondy, J. A. and U. S. R. Murty, *Graph Theory with Applications*, North Holland, 1976.

[Chan81]
Chang, C. L., "On Evaluation of Queries Containing Derived Relations in a Relational Data Base", in *Advances in Data Base Theory* **Vol. 1**, edited by H. Galaire, J. Minker and J. M. Nicolas, Plenum Press, New York, N.Y., 1981, pages 235-260.

[Codd70]
Codd, E. F., "A Relational Model of Data for Large Shared Data Banks", *CACM* **13**, 6 (1970), pages 377-387.

[Date82]
Date, C. J., *An Introduction to Database Systems, 3rd edit.*, Addison-Wesley, Reading, MA, 1982.

[Ende72]
Enderton, H. B., *A Mathematical Introduction to Logic*, Academic Press, New York, N.Y., 1972.

[Gall78]
Gallaire, H. and J. Minker, *Logic and Data Bases*, Plenum Press, New York, N.Y., 1978.

[Gall81]
Gallaire, H., J. Minker, and J. M. Nicolas, *Advances in Data Base Theory, Vol. 1*, Plenum Press, New York, N.Y., 1981.

[Gall84]
Gallaire, H., J. Minker, and J. M. Nicolas, "Logic and Databases: A Deductive Approach", *ACM Computing Surveys* **16**, 2 (June 1984).

[Ioan85]
Ioannidis, Y. E., *Bounded Recursion in Deductive Databases*, Memorandum No. UCB/ERL M85/6, University of California, Berkeley, 1985.

[Naqv84]
Naqvi, S. and L. Henschen, "On Compiling Queries in Recursive First-Order Databases", *JACM* **31**, 1 (January 1984).

[Reit78]
Reiter, R., "Deductive Question-Answering on Relational Data Bases", in *"Logic and Data Bases"*, edited by H. Galaire and J. Minker, Plenum Press, New York, N.Y., 1978, pages 149-177.

[Robi65]
Robinson, J. A., "A Machine Oriented Logic Based on the Resolution Principle", *JACM* **12**, 1 (January 1965), pages 23-41.