A Reconsideration of the Termination Conditions of the Henschen-Naqvi Technique

David A. Briggs

17 September 1986

COINS Technical Report 87-11

Abstract

Henschen and Naqvi describe a sophisticated technique for translating queries on recursively defined relations of a Horn clause data base into iterative programs that invoke a query processor for conventional select-project-join queries of the relational algebra. In a recent comparison of methods for evaluating recursive queries the Henschen-Naqvi technique is cited as one of the most efficient currently available. Nevertheless, the technique is flawed in that there exist recursive queries for which the answer computed from the translated program will be different from the correct answer under the usual semantics for Horn clause data bases. The problem is that the termination conditions as given are premature; the iteration may stop before all of the answers have been obtained. In this paper we review the Henschen-Naqvi technique, exhibit a recursive query and database instances for which the technique fails, note why in general the termination condition is premature, and describe a condition sufficient to guarantee that all answers are obtained. We also offer a variation of their algorithm that should be more efficient for some data base instances.

In [HENS84] the authors describe a sophisticated technique for translating queries on recursively defined relations of a Horn clause data base into iterative programs that invoke a query processor for conventional select-project-join queries of the relational algebra. In a recent comparison of methods for evaluating recursive queries ([BANC86]), the Henschen-Naqvi technique is cited as one of the most efficient currently available, and this author admires the insight that led to the method. Nevertheless, the technique as presented in [HENS84] is flawed in that there exist recursive queries for which the answer computed from the translated program will be different from the correct answer under the usual semantics for Horn clause data bases (for a description of those semantics, see [LLOY84]). The problem is that the termination conditions as given are premature; the iteration may stop before all of the answers have been obtained. In this paper we review the Henschen-Naqvi technique, exhibit a recursive query and database instances for which the technique fails, note why in general the termination condition is premature, and describe a condition sufficient to guarantee that all answers are obtained.

In the interest of clarity much of the terminology and discussion presented in [HENS84] will be repeated here, with occasional amplification or shifts in representation. The paper presumes in the reader some familiarity with logic, logic programming, and the relational model of data; for example, unification and resolution are not explained here.

1 Preliminaries : Logic, Data Bases, and Recursion

A Horn database (without function symbols) consists of a finite number of clauses, each with exactly one positive literal and zero or more negative literals. A clause with zero negative literals is termed a unit clause, and if all of its arguments are constants rather than variables, then it is a ground positive literal, or ground positive unit.[1] For each distinct predicate (we will use 'predicate' and 'relation' as synonyms), a ground positive literal of that predicate asserts that the tuple of constants is in its relation, and we will imagine that all ground units for a particular predicate are stored as in a conventional relational data base. The collection of all relations with ground units will be collectively referred to as the *extensional data base*, or *EDB*, and a particular relation that has a ground unit will be regarded as

[1]Clauses whose positive literals contain variable arguments not mentioned in any of the negative literals, and this includes non-ground unit clauses as a special case, complicate the semantics, so we will assume that no clauses of that ilk are allowed.

an *EDB relation.*

A clause that contains some negative literals is a rule that indicates a way to infer that a tuple is a member of the relation for the positive literal. For example, the clause

```
uncle(x,y), ¬brother(x,z), ¬parent(z,y)
```

under the appropriate interpretation of the components, can be seen to express the rule

```
    For any x and y, if there is a z such that x is z's
brother and z is y's parent, then x is y's uncle.
```

Alternatively, the rule can be viewed as asserting that the uncle relation is a superset of a select–project–join expression involving the brother and parent relations, the negative literals mentioned in the clause. The rules will be collectively referred to as the ***intensional data base***, or *IDB*, and any predicate that occurs as the positive literal in a rule (also termed the ***head*** of the rule), is regarded as an *IDB predicate.*

A particular predicate may be an IDB predicate, an EDB predicate, or both an IDB and an EDB predicate, depending upon the clauses in which it appears as the positive literal. It is always possible to construct an equivalent Horn data base in which the IDB and EDB predicates are disjoint (see [BANC86] for details), so for simplicity we will assume they are disjoint in our data base. Now, the extensions of the EDB predicates are just the tuples asserted to be in them by the ground units. The extensions of the IDB predicates are those tuples that can be inferred to be in them from the contents of the EDB relations and the rules. In the absence of function symbols the extensions of the IDB predicates will be finite and computable (see [LLOY84] for details).

A query on a Horn clause database is a set of negative literals, and can always be expressed as a select–project–join expression[2] over the extensions of the relations of the literals. For example, the query

[2]In fact, the negative literals define a select–join expression, but it may be that some extraneous variables have been introduced by rule applications, and they will be projected away. For example, in answering the query ¬uncle('Fred',x), if the earlier rule is used, the subquery ¬brother('Fred',z) ¬parent(z,y) contributes to the answer, but the values for z are not of interest, and will be projected out.

```
¬parent(x, y), ¬parent('Mary', y), ¬parent(w, x)
```

when projected on `w` will obtain the grandparents of Mary's children.

The problem in evaluating queries is that the IDB extensions are not explicitly stored, and queries against an IDB relation must therefore compute enough of its extension to guarantee that all the answers are obtained that would be obtained if its extension were fully computed. In [REIT78] a method is given to translate queries involving IDB relations into relational algebra queries involving EDB relations, but the method fails for a relation that is 'recursive', loosely, a relation whose definition involves itself. The method of [HENS84] was offered as an extension of Reiter's work to handle queries involving recursively defined relations. Before going into the details of the method, one should recall that the overall goal is to determine, for queries against recursively defined relations, a program that can be run against the EDB and obtain the same results that would be obtained if the IDB relation were fully computed and added to the data base as an explicit EDB relation. In this sense, the rules that determine the extensions of the IDB relations are "compiled away", and the overhead of rule application avoided.

Following the Henschen-Naqvi paper, we will represent the rules of the database in a Connection Graph, with a few changes for clarity. Each rule will have a node in the graph. The nodes of the graph are complex in that they have identifiable parts, viz., the literals of the rule. Arcs between nodes will specify not just the nodes at the head and tail of the arc, but the particular literals within the nodes. We include in the graph a directed arc from a negative literal $\neg P(\ldots)$ of node n to a literal $P(\ldots)$ of node m, for P any predicate, if the two literals unify (this immediately forces that the only literal that can be the target of an arc is the head of the clause represented by node). This characterization is slightly different from the one given in the paper, but is readily grasped and makes the next definition simpler.

Given a connection graph for a database, a ***potential recursive loop (PRL)***, is a cycle of edges $(e_1, \ldots, e_n)$ in the graph such that

1. The substitutions required to unify the literals that occur at the heads and tails of all of the edges are consistent with one another, that is, one could unify around the cycle.

2. If one resolves the clauses at the tails of the arcs e_1 through e_n, resolving on the literals suggested by the arcs, and then resolves in a distinct instance of the clause at the tail of arc e_1, resolving on the literals suggested by e_n, the result is not merely a renaming of that clause. This condition ensures that resolving around the loop changes the clause in some way.

3. The resolution of any subset of the clauses included in the cycle does not yield a tautology.

The conditions identifying PRL's are precisely those for identifying relations whose extensions are recursively defined, since traversing an arc corresponds to applying the rule at the head of the arc, and in a PRL it is apparently possible for a rule contributing to the extension of a relation to ultimately invoke itself.

Consider Figure 1, adapted from [HENS84], of a PRL as it would be used in evaluating a query against a recursively defined relation. The following terminology will be used here, as in their paper.

The *query literal*, or *goal*, is the negative literal that attaches to the loop. It may have some of its arguments bound to constants, but it must be unifiable with the head of the clause to which it attaches. In our diagram, the query literal is the $\neg D$ node.

The *start literal/start clause* is the literal/clause to which the query literal attaches. In the diagram, the start literal is D'_0 and the start clause is the clause it belongs to.

The literal/clause within the loop whose successor is the start literal is the *end literal/end clause.* $\neg D_n$ is the end literal of the diagram.

The *closing edge* is the arc that connects the end literal to the start literal.

The literals $\neg D_i$ and D'_i at the tail and head of all arcs other than the closing edge are called *cycle literals.*

An edge leading from a cycle literal out of the PRL is called an *exit edge*, for example, the edge from $\neg D_1$ to D''_1 is an exit edge.

The *augmented loop residue (ALR)* is the resolvent obtained by resolving the start clause with its successor clauses in the cycle, up to and including the end clause. For the loop of the example, $D'_0, \neg E_1, \neg E_2, \ldots, \neg E_n, \neg D_n$ is the ALR.

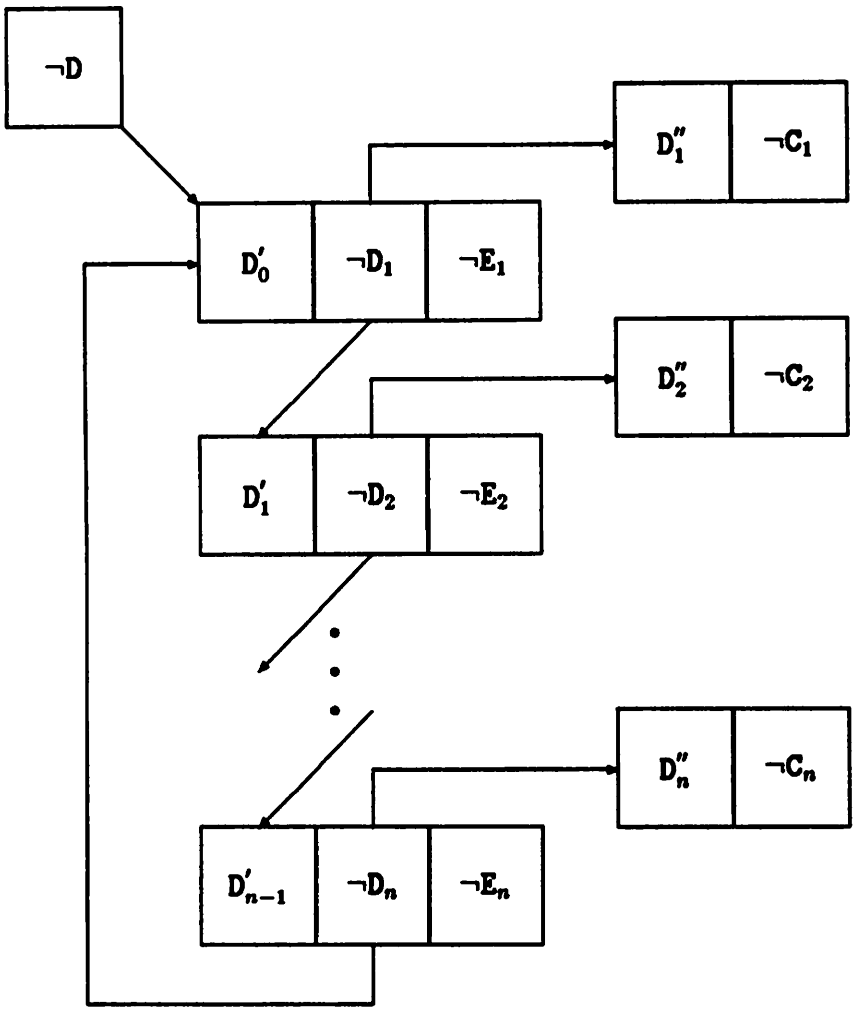

Figure 1: A PRL with attached query literal, ¬D

An *augmented exit expression (AEE)* is any resolvent obtained by resolving the start clause up to some clause containing an exit edge, and then resolving out the edge. The AEE's of the example are $D'_0, \neg E_1, \neg C_1$ and $D'_0, \neg E_1, \neg E_2, \neg C_2$, and so forth.

The ALR will have the form

`P(...)`, some negative literals, `¬P(...)`

where P is the predicate of the start literal, and under the conditions of the PRL, it is possible to resolve the ALR with a distinct instance of itself by unifying the end literal with the start literal of the distinct instance. The resulting clause will still contain instances of the start and end literals, and another instance of the ALR could be resolved with it. It may be possible to be repeat this process indefinitely.

Resolving the query literal with any of the resolvents obtained from the above process will remove the start literal and may provide constant bindings for some of the variables, but the result is still not evaluable against the EDB because it contains an IDB relation, namely the end literal. If we resolve the end literal with an AEE, however, we remove it from the clause, and assuming for the sake of the discussion that the cycle literals and start literals are the only IDB relations, we have a select–project–join expression against the EDB whose result contributes to the answer for the query. The complete answer is the union, over all expressions obtainable by resolving around the loop zero or more times and out an exit, of the evaluation of the expression against the EDB.

2 The Henschen-Naqvi Method

The key insight of the Henschen-Naqvi technique is that under certain conditions, knowledge of which of the query literal's arguments will be bound to constants can be profitably used in evaluating these expressions. For each distinct specification of bound variables in the query literal, their technique derives a distinct program that is tailored for that pattern. Of course, for an n-argument relation, this means 2^n distinct programs, so one hopes for small n, but the idea is really quite clever. To fully explicate it, we need some more definitions.

A *query form* is a literal in which each argument is either a variable or a dummy constant. An *instance* of a query form is a the query form with the dummy constants replaced by actual constants.

For a particular query form of the start literal, the ***determined variables*** of the ALR are defined as follows.

1. The variables of the start literal in the positions of the dummy constants of the query form are determined variables.

2. If x is a determined variable, and ¬L is a literal of the ALR other than the end literal, and ¬L has x as an argument, then any variable y occurring in ¬L is also a determined variable.

The literals of the ALR containing determined variables, excluding the start and end literals, are called the ***determined part*** of the ALR.

The literals of the ALR other than the determined literals and the start and end literals are called the ***induced part*** of the ALR.

Intuitively, the determined part of the ALR is the largest subclause that allows the selection conditions, the constants, to propagate through what will be EDB relations, under the assumptions we will make. The selection conditions inhibit the size of the join results, but they cannot be propagated through the end literal, because its extension is not explicitly stored. Also, under the simpliflying assumptions we will make, evaluation of the determined parts of successive answer expressions will be subject to a significant economy.

By rearranging the literals of the ALR, it can be expressed as four clauses : `Q`, `¬A`, `¬Q`, and `¬B`, where `Q` is the start literal, `¬A` is the induced part, `¬Q` is the end literal, and `¬B` is the determined part. To precisely characterize the resolvents we would obtain by resolving around the loop and out an exit, we need to identify the overlap of variables among these parts, so we define the following sets of variables.

Z the set of determined variables of the ALR

Z′ the determined variables occurring in the start literal

Z″ the determined variables occurring in the end literal

W the set of non-determined variables of the ALR

$\mathbf{W}'$ the set of non-determined variables of the start literal

$\mathbf{W}''$ the set of non-determined variables of the end literal

$\mathbf{X} = Z - Z'$

$\mathbf{Y} = W - W'$

Employing these terms, the variables occurring in the parts of the ALR can be identified as follows[3]

$$Q(W', Z'),\ \neg A(Y, W'),\ \neg Q(W'', Z''),\ \neg B(X, Z')$$

An augmented exit expression can be written as

$$Q(W', Z'),\ \neg E(V, W', Z')$$

where V are variables of the exit expression that are not in W' or Z'.

In the discussion that follows, we make all of the simplifying assumptions stated on pps. 62-63 of the original paper and repeated here, and one final assumption not mentioned there.

1. There is only one exit edge from the PRL.

2. Except for the start and end literals, the ALR consists entirely of EDB literals.

3. There is no proper subset of the literals of the ALR which has no variables in common with the remaining literals of the ALR.

4. There is only one PRL in the connection graph.

[3]This notation suggests that all of the variables of Z' appear in the determined part, and that all of the variables of W' appear in the induced part. If this is not true one can imagine them as added arguments whose bindings are irrelevant to the truth value of the subclause. This tack is theoretically sound, but to materialize the relation of the augmented subclause one would have to take the Cartesian product of the set of tuples satisfying the original clause with all constants mentioned in the data base, as many times as there are missing variables. There are other, less costly measures that are difficult to describe briefly.

5. The unifiers of the arcs in the PRL do not force any of the variables of the start literal to be constants, nor do they equate distinct variables of the start literal.

6. The determined variables of the start and end literals are equal in number and occupy the same positions.

7. The end literal is a renaming of the start literal.

In a nutshell, the assumptions constrain the set of expressions to be evaluated against the EDB to be a particularly tractable one, and as the authors state, most can be relaxed without difficulty.

We want to characterize the expressions evaluable against the EDB that might produce answers. As mentioned above, such an expression is obtained by resolving around the loop zero or more times and out the exit, and we can describe them by specifying how we will perform the resolutions of the ALR or AEE with the end literal. In resolving in additional instances of the ALR (or AEE, as the case may be), there must be no overlap of its variables and those of the clause with which it is being resolved. We will effect this by maintaining superscripts on the variables. All the variables of the ALR or AEE will be superscripted by 1. When we prepare to resolve in another instance of the ALR, or the AEE to leave the loop, we will increment the superscripts of the variables in the resolvent obtained so far and unify the start literal of the ALR or AEE with the end literal of the result. In computing the unifier, we will always replace variables of the start literal of the ALR with variables of the end literal of the other clause, so the variables of the other clause will not be affected by the resolution. To completely characterize the resolvents we need to consider the unification of the start literal of the ALR with the end literal. Supposing that distinct variables of the ALR are distinguished by subscripts, that is, all are of the form x_i^1 for $i \epsilon \{1, \ldots, | \mathbf{W} \cup \mathbf{Z} |\}$, and placing the end literal above the start literal matches them as follows

$$\begin{array}{r} \neg \mathtt{Q}(x_{k_1}^1, x_{k_2}^1, \ldots, x_{k_n}^1) \\ \mathtt{Q}(x_{j_1}^1, x_{j_2}^1, \ldots, x_{j_n}^1) \end{array}$$

define a mapping $\theta : \{1, \ldots, |\mathtt{W} \cup \mathtt{Z}|\} \longrightarrow \{1, \ldots, |\mathtt{W} \cup \mathtt{Z}|\}$ as

$$\theta(i) = \begin{cases} k_m & \text{if } i = j_m \text{ for some } m \\ i & \text{else} \end{cases}$$

Now we can inductively define the substitution used to resolve in the ith instance of the ALR, σ_i, as follows,

$$\sigma_1 = \text{identity substitution}$$
$$\sigma_{i+1}(x_j^1) = \begin{cases} x_j^1 & \text{if } x_j^1 \text{ is not in } \mathtt{W}' \cup \mathtt{Z}' \\ [\sigma_i(x_{\theta(j)}^1)]^{(+1)} & \text{else} \end{cases}$$

where the superscript operator $(+n)$ means that all variables mentioned in the expression should have their superscripts incremented by n. In words, the $i+1$st substitution makes no change to a variable that does not occur in the start literal, and replaces a variable that does occur in the start literal with its correspondent in the end literal of the clause obtained so far. That end literal was derived from the most recent ALR to have been resolved into the expression, so the variable in that position will be the result of incrementing the superscript of the result of applying the previous substitution to the original variable in that position, $x_{\theta(j)}^1$.

Employing this notation, the resolvent obtained by circling the loop i times and out the exit, $\mathtt{RES}_i$, can be expressed as follows

$$[\mathtt{Q}(\mathtt{W}', \mathtt{Z}')]^{(+i)} \vee \bigvee_{j=2}^{i+1} [\sigma_{i+2-j}(\neg\mathtt{A}(\mathtt{Y}, \mathtt{W}'))]^{(+(j-1))} \vee$$
$$\sigma_{i+1}(\neg\mathtt{E}(\mathtt{V}, \mathtt{W}', \mathtt{Z}')) \vee \bigvee_{j=2}^{i+1} [\sigma_{i+2-j}(\neg\mathtt{B}(\mathtt{X}, \mathtt{Z}'))]^{(+(j-1))}$$

When this expression is resolved with the query form instance, the $(\mathtt{Z}')^{(+i)}$ variables will be bound to the specific constants, and the variables of $(\mathtt{W}')^{(+i)}$ will be projected on to obtain the answer tuples.

For convenience, let's identify a particular instance of $\neg\mathtt{A}$ or $\neg\mathtt{B}$ within such a resolvent by the value of the index j. Ignoring the start literal, which will vanish when the query literal is resolved with the expression, we list the literals in the suggestive pattern

$$\neg\mathtt{A}_{i+1}, \neg\mathtt{A}_i, \ldots, \neg\mathtt{A}_2, \neg\mathtt{E}, \neg\mathtt{B}_2, \ldots, \neg\mathtt{B}_i, \neg\mathtt{B}_{i+1}$$

Matching subscripts for a pair of clauses $\neg A_j$ and B_j indicate that they were derived from the same instance of the ALR, the $(i+2-j)$th instance, and later instances will be nearer the $\neg E$ clause, which forms a bridge between the determined and induced part of the expression. The determined variables of the start literal, $(Z')^{(+i)}$, will occur in $\neg B_{i+1}$, and perhaps in other instances of the determined part. The non-determined variables of the start literal, $(W')^{(+i)}$, will occur in $\neg A_{i+1}$, and perhaps other instances of the induced part. Now, under the simplifying assumptions and using the formal notation detailed above, it can be shown that the pattern of overlap between $\neg A_j$ and $\neg A_{j+1}$ is the same regardless of the resolvent i or the value of j. In other words, if we were to place successive instances of the $\neg A$ clause in a column

$$\begin{array}{l} \neg A_{j+1}(\mu_1, \mu_2, \ldots, \mu_n) \\ \neg A_j \ \ (\nu_1, \nu_2, \ldots, \nu_n) \\ \neg A_{j-1}(v_1, v_2, \ldots, v_n) \end{array}$$

we would observe that if μ_i and ν_k were the same variable, then ν_i and v_k would also be the same variable. The same remark holds for the instances of the determined part within a particular resolvent, and for the overlap of the $\neg E$ clause with its "nearest" $\neg A$ and $\neg B$ clauses, $\neg A_2$ and $\neg B_2$. The regularity of the answer producing resolvents has a number of consequences that can be exploited in an iterative computation. First, note that the determined part of resolvent RES_i is a subclause, up to renaming, of the determined part of RES_{i+1}. Since the constants of the query are fixed for all the resolvents, in evaluating the determined part of RES_{i+1}, results from evaluating the determined part of the previous resolvent can be used, rather than evaluating the whole determined part from scratch, surely a savings.

Moreover, the remainder of the expression that will be evaluated against the EDB also exhibits a predictable pattern for successive resolvents, although the description of its computation given in the paper is in error[4].

In the discussion that follows, we would like to regard the determined part, induced part, and exit expression as single relations of the EDB rather

[4]If the non-determined variables of the start and end literals are not disjoint, modifying what Henschen and Naqvi term the *G* expression from one iteration to the next may require more subsitutions than their algorithm indicates. More than one literal of the clause may be affected.

than clauses. Since they are presumed to consist entirely of EDB predicates, they could in theory be computed from their constituents and stored separately.

Given that the pattern of variable overlap between neighbor instances of the determined or induced parts is the same for all resolvents and for all neighbors, we can view the instances as computing functions from tuples provided as input to tuples produced as output. We describe the function for the instances of the determined part. Each input tuple provides bindings for certain of the components of the B relation, precisely those components which would overlap with the next higher indexed instance. The input tuple thus provides a selection condition on the B relation. The tuples which satisfy the selection condition are then projected on the components which overlap with the next lower indexed instance of the clause, and this is the output for the single input tuple. The output for a set of tuples as input is the union over all of the input tuples of the result each would produce.

If we dub this function f, then $\mathbf{RES}_i$ will provide the determined variables of the $\neg$E clause with bindings from $f^i(\{\vec{c}\})$ where $\vec{c}$ is the tuple of constants from the query form instance.

We can similarly regard the exit expression instance for $\mathbf{RES}_i$ as a function, fixed for all i, on the tuples it receives to a set of tuples to be used for the induced part, and each of the $\neg$A instances as functions from sets of tuples to sets of tuples. The exact nature of these functions is determined by the pattern of overlap and the particular extensions of the A, B, and E relations, but the ith resolvent can be viewed as a $2i+1$ stage pipeline composed of i B units, followed by a single E unit and i A units (see Figure 2).

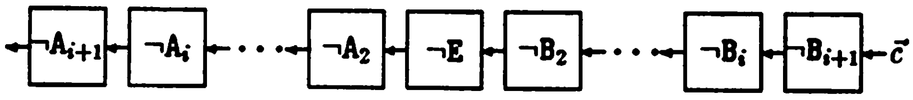

Figure 2: Pipeline representation of $\mathbf{RES}_i$ with input $\vec{c}$

Now one can readily see how the economy of computation of the determined part is justified. From one resolvent to the next, we are inserting an additional B unit into the functional pipe before the E component, but the results from the earlier stages are not affected, since the input from the query form instance has not changed, and the functions are constant regardless of resolvent index.

The exit expression and induced parts do not exhibit a similar economy because although the functional components of the pipe are the same from one to the next except for the addition of another A unit, the input to the pipe at successive iterations may be different. If, however, a value d fed in at iteration i reappears at iteration $i+k$, then the results obtained from d at iteration i could be fed through k more A units to obtain d's contribution at iteration $i+k$. This fact is noted by the authors when they describe the data structures for detecting cycles, and will be discussed again below.

It remains to consider termination conditions. It is impossible to determine the number of iterations necessary independently of the extensions of the relations, so we must look for some behavior during successive iterations which implies that further iterations will yield no new answers. The most obvious condition is that the B pipe "runs dry", that is, at some iteration i, $f^i(\{\vec{c}\}) = \emptyset$. A moment's consideration convinces one that given the nature of the functions computed by these components, no more answers will be produced.

There is no guarantee, however, that the B pipe will run dry and Henschen and Naqvi suggest that the program look for cyclic behavior in the answers it produces to ascertain that the iteration can stop. Their description of exactly what cyclic behavior is sufficient grounds for termination is vague, and in our reading, incorrect. We will describe their scheme, exhibit cases where it fails, and discuss why it fails.

Following Henschen and Naqvi, we will call the tuples presented to the E unit by the B units of the pipeline at each iteration of the loop *driver tuples*. Since the B relation is presumed finite, if driver tuples are produced at every iteration, some must be repeated. The authors suggest that the program maintain a tree for each driver tuple that is produced. The tree will have as its root its associated driver tuple, and for each iteration in which that driver tuple is produced by the B pipeline, the tree will have a level recording the yield from the E–A units of the pipe for that iteration if given that driver tuple as input. The tree is arranged to facilitate the computation of answers as follows. Suppose driver tuple d first appears at iteration i and produces answers $a_1, \ldots, a_n$. The tree for d is created with the root d, and first generation children (a_j, i) for all $j \epsilon \{1, \ldots, n\}$. If d is next produced as a driver at iteration $i + k$, instead of feeding d into the pipe consisting of E and $i + k$ A's, one may feed the a_j's into a pipe consisting of k A's and obtain the same results with some savings. If doing this for a specific a_j yields some answers, then those answers are added to the tree as children of a_j, along with the iteration number $i + k$. If an a_j fails to produce any answers, its node may be marked as a dead end, and ignored in subsequent iterations that turn up driver d. Again, this is justified by the fact that if a k length pipe of A's fails to produce an answer when fed a_j, any longer pipe would also fail to produce an answer, so the node (a_j, i) can make no further contributions to the set of answers.

The previous remarks indicate one condition under which a leaf of the tree may be marked as a dead end, but this condition by itself is not sufficient to ensure that ultimately all leaves of the tree will be marked, since there may be cyclic behavior in the answers produced as well. The paper does not describe in general terms how a cycle of answers is recognized, but from its example it appears that if an answer tuple a is produced from driver d at iteration i, so that (a, i) is a node in d's tree, then if at a subsequent iteration, $i + k$ say, d is produced as a driver and the answer a recurs as a descendent of node (a, i), the node $(a, i + k)$ is added to the tree, but marked as dead. Presumably any answers it would generate would be obtainable from node (a, i). If all the leaves of a driver tuple's tree have been marked as dead ends, then that driver can be ignored in future iterations. Processing can terminate at the first iteration which turns up no new drivers with all the old drivers marked dead. Clearly, this is guaranteed to eventually occur, since the relations are finite. That this condition stops the iteration too quickly can be seen from the following examples.

```
RULES
Q(x,y)   ¬EQ(x,y)
Q(x,y)   ¬B(x,z) ¬Q(z,w) ¬A(w,y)

EXTENSIONS
```

$$\mathbf{A} \;=\; \{(\mathrm{p},\mathrm{b}),(\mathrm{b},\mathrm{c}),(\mathrm{c},\mathrm{p})\}$$

$$\begin{aligned}\texttt{B} &= \{(\texttt{e},\texttt{f}),(\texttt{e},\texttt{a}),(\texttt{f},\texttt{g}),(\texttt{g},\texttt{h}),\\ &\quad(\texttt{h},\texttt{a}),(\texttt{a},\texttt{p}),(\texttt{p},\texttt{q}),(\texttt{q},\texttt{r}),(\texttt{r},\texttt{a})\}\\ \texttt{EQ} &= \{(x,x) \mid x \text{ is a constant mentioned in A or B}\}\end{aligned}$$

Consider the processing of the query form instance `¬Q('e',?)`.

From resolving the query against the first rule, we would obtain the answer `e`. The exit expression with the determined variable bound to the input `e` is `¬B('e', z) ¬EQ(z, w) ¬A(w, y)` with `y` the variable to be projected on. It produces no answers.

The following table shows the driver tuples produced at each iteration of the loop, the answer sequences that would be stored in the tree for each driver, and points at which sequences and driver tuples are marked dead.

Iterations	Drivers	Sequences
1	f	null – mark f as dead
	a	(p,1)
2	p	null – mark p as dead
	g	null – mark g dead
3	h	null – mark h as dead
	q	null – mark q as dead
4	a	(p,1) (p,4) – a cycle, so mark the sequence as dead; since this is the only sequence for a, mark a dead
	r	null – mark r as dead
5	a	already dead
	p	already dead

Since at iteration 5 we produced no new driver tuples, and all those produced have been marked dead, the processing would stop at this point. However, **a**'s tree was incorrectly marked dead. At iteration 5, **a** should produce the answer **b**, and at iteration 9, **a** would again be produced as a driver, and should yield the answer c. Another data base instance for which the program would fail, for a somewhat different reason, is

$$\begin{aligned}
\mathtt{A} &= \{(\mathtt{b3},\mathtt{c1}),(\mathtt{c1},\mathtt{c2}),(\mathtt{c2},\mathtt{c3}),(\mathtt{c3},\mathtt{c4}),(\mathtt{c4},\mathtt{c5}),\\
&\quad (\mathtt{c5},\mathtt{c6}),(\mathtt{c6},\mathtt{c1}),(\mathtt{c2},\mathtt{c7}),(\mathtt{c7},\mathtt{c8}),(\mathtt{c8},\mathtt{c9}),\\
&\quad (\mathtt{c9},\mathtt{c10}),(\mathtt{c10},\mathtt{c11}),(\mathtt{c11},\mathtt{c12}),(\mathtt{c12},\mathtt{c13})\}\\
\mathtt{B} &= \{(\mathtt{e},\mathtt{b1}),(\mathtt{e},\mathtt{b2}),(\mathtt{b1},\mathtt{b3}),(\mathtt{b2},\mathtt{b1}),(\mathtt{b2},\mathtt{b4}),\\
&\quad (\mathtt{b4},\mathtt{b5}),(\mathtt{b5},\mathtt{b6}),(\mathtt{b6},\mathtt{b7}),(\mathtt{b7},\mathtt{b1}),(\mathtt{b7},\mathtt{b2})\}\\
\mathtt{EQ} &= \{(x,x) \mid x \text{ is a constant mentioned in A or B }\}
\end{aligned}$$

Again, the answer e is produced from the first rule. Evaluating the exit expression produces no answers. The table is as follows.

Iterations	Drivers	Sequences
1	b1	(c2, 1)
	b2	null – mark b2 as dead
2	b1	(c2, 1) (c3, 2)
		(c2, 1) (c7, 2)
	b3	null – mark as dead
	b4	null – mark as dead
3	b3	already dead
	b5	null – mark as dead
4	b6	null - mark as dead
5	b7	null - mark as dead
6	b1	(c2, 1) (c3, 2) (c1, 6)
		(c2, 1) (c7, 2) (c11, 6)
	b2	already dead
7	b1	(c2, 1) (c3, 2) (c1, 6) (c2, 7)

		a cycle, so mark the sequence dead (c2, 1) (c7, 2) (c11, 6) (c12, 7)
	b3	already dead
	b4	already dead
8	b3	already dead
	b5	already dead
9	b6	already dead
10	b7	already dead
11	b1	(c2, 1) (c7, 2) (c11, 6) (c12, 7) produces no new answers, so it can be marked as dead. Thus, b1 can be marked as dead.
	b2	already dead
12	b1	already dead
	b3	already dead
	b4	already dead

Since this iteration produced no new drivers, and all drivers have been marked dead, processing would halt. The first sequence was prematurely marked dead at iteration 7, and should have gone on to yield additional answers `c4, c5, c6 c8, c9, c10,` and `c13`.

3 Correct Termination Conditions for the Cyclic Case

To elucidate the problem we shall shift to a graphical representation of the computation. Construct a graph G_B with a node n_S for each $S \subseteq$ B, and a distinguished node, n_A. Draw a directed edge from n_S to n_T, for $S, T \subseteq$ B, if the projection of the tuples of S on the components that a B pipe unit projects on, when fed into a B pipe unit, would select the tuples of T. Draw an arc from n_A to the unique node n_S that corresponds to the subset of B that would be selected by the vector of constants of the query form instance. In effect, the graph mimics the behavior of the of the B pipe unit, but carries along the components of B that would be projected away. Evidently, the driver tuples produced at the ith iteration would be derived from the set S of the unique node i arcs from n_A. We will denote the subset of B reached at the ith iteration by D_i.

Since B is finite, this graph must also be finite, and since every node has a unique successor, at some point the sequence of driver tuples must cycle, with some period p[5]. Let $D_l, D_{l+1}, \ldots, D_{l+p} = D_l$ be the cycle, with l and p the least values that determine a cycle. The tuples that will be repeatedly presented to the non-determined part of the pipeline will be in

$$\bigcup_{i=l}^{l+p-1} D_i = \mathcal{D}$$

For a particular tuple d in $\mathcal{D}$, there is nothing to prevent d from occurring in several of the cycle D_i's, so for each d we define the *cycle set* for d, $K_d = \{i\epsilon\{0,\ldots,p-1\} \mid d\epsilon D_{l+i}\}$ Now, future iterations at which d is presented to the non-determined part of the pipeline have a precise characterization, specifically,

$$\{l + mp + k \mid m\epsilon\omega \text{ and } k\epsilon K_d\}$$

For each driver tuple d produced, define the *occurrence set* for that tuple, O_d, to be the set of iterations at which it is produced as a driver. Evidently the occurrence set of a driver tuple will either be finite, if the tuple is not in $\mathcal{D}$, or will consist of the a set describable as above and perhaps finitely many other numbers, if the tuple is contained in some of the driver sets *before* the cycle.

Our second graph G_A has a node for each tuple in A + B, the disjoint union of A and B. For a node n_b derived from some $b\epsilon$B, draw directed arcs to nodes n_a derived from $a\epsilon$A, if tuple a would be selected by an A unit when given the result of an E unit fed the appropriate components of b. Draw an arc from node n_{a_1} to node n_{a_2}, derived from $a_1, a_2\epsilon$A, if a_2 would be selected by an A unit fed the appropriate components of a_1. Again, the graph mimics the action of the pipeline, although in this graph the action

[5] If the pipe runs dry, it cycles on $n_\emptyset$.

is recorded at the individual tuple level, rather than at the subset level, and so some nodes may have no successors and some may have many successors.

Now, the answers produced at iteration i are the tuples in the union over all d in D_i of the tuples of **A** corresponding to the nodes of G_A reachable from n_d in exactly i arcs, projected onto the components corresponding to the non-determined variables of the start literal.

Henschen and Naqvi's suggestion for recording a tree for each driver tuple represents part of the G_A graph. A typical node in their tree, (a, i), indicates that the answer tuple a is i arcs from the node for the driver tuple at the root of the tree. Ancestors of this node in the tree identify other nodes along that length i path. The only nodes of the paths that are explicitly recorded in the tree, however, are those that are at distances from the driver tuple node which are also in its occurrence set. To correctly terminate the algorithm, we need a condition for marking a leaf node dead which will not lose any answers, and which is also certain to eventually become true. A sufficient condition for safely marking the node (a, i) dead is that for all nodes b of the underlying graph reachable from a in k arcs with $(i + k)\epsilon O_d$, where d is the driver tuple root of the tree, either

1. b has already been recorded as an answer, or
2. there is another leaf node (c, i), say, with b reachable from c in r arcs, and $(i + r)\epsilon O_d$ and $r < k$

The examples illustrate that correct termination depends upon an accurate description of occurrence sets for the driver tuples. Drawing the graphs for the first example, one can see that the second occurrence of the driver tuple **a** is not within the cycle of driver sets, which begins at the next iteration (see Figure 3). Its period is relatively prime to the period of the cycle in the second graph, and so ultimately all of the nodes of the second cycle should produce answers.

In the second example the driver tuple **b1** is in two of the sets of the cycle, so the first repetition of the answer is not actually representative of its future cyclic behavior.

The correct test for marking a leaf node dead is not only that the leaf node repeats an answer given by an ancestor, but also that the iterations of the ancestor and the leaf correspond to the same set within the driver set cycle. More specifically, it is sufficient that the two iterations are congruent (mod p), where p is the period of the driver set cycle, provided that the ancestor was produced after the cycle had been initiated. In fact, if K_d consists of the union of some of the cosets of the cyclic subgroup of Z_p generated by some $i \epsilon Z_p$, it is sufficient that the iterations be equivalent (mod i). Under those conditions the cycle from a node to itself revealed in the tree is an integer multiple of the period of the cycle of the driver sets, and as such, it is extraneous to the computation of answers. If, in G_A, a is a node that lies on a length i path from the driver tuple node n_d that includes the cycle, with $i \epsilon O_d$, then there is a shorter path that doesn't use the cycle, of length j, with $j \epsilon O_d$. If K_d consists of the union of some of the cosets of the cyclic subgroup of Z_p, generated by some i in Z_p then, in effect, the driver d cycles on a smaller period within the larger period, and the smaller period can be used in the test. Since the sets are finite, this test must eventually become true on each path.

This test requires determining values for l, p, and perhaps K_d, and so the algorithm must be augmented to check for a cycle in the driver sets. It can be shown that if $D_{i+k} \subseteq D_i$, for some $i, k > 0$, then

1. For all $j \geq i + k, D_j \subseteq D_{j-k}$

2. The period p of the driver set cycle will be a divisor of k

A plausible strategy for detecting the driver set cycle is to check the current set for inclusion in previous sets, to determine the i and k mentioned above. If an accurate accounting of the cardinalities is maintained, a simple cardinality check may rule out a lot of possibilities. Once an i and k for which $D_{i+k} \subseteq D_i$ have been found, keep track of the cardinalities of every set and its kth successor. As soon as one of them holds equality with its predecessor, the cycle has been entered. It still remains to determine if the actual cycle is a divisor of k. Again, cardinality checks may rule out some possibilities without the expense of comparing the sets for equality, or the smallest set of the cycle may be chosen as the guinea pig to be compared with the sets that are a divisor's distance from it in the list.

Given that correct termination in the presence of cycles requires determining the period of the cycle of driver tuple sets, we offer the following variation of the Henschen-Naqvi algorithm, which in effect merges the information recorded in the separate driver tuple trees into one structure, and may lead to earlier termination.

Before computing any answer tuples, we identify all driver tuples and their associated occurrence sets. As noted above, these sets consist of a finite part and, if the tuple is one of the cycling drivers, an infinite part that can be represented by the subset of Z_p, where p is the period of the cycle, specifying the sets of the cycle that contain it as an element.

With the drivers and occurrence sets in hand, we next walk through the G_A graph in a wave front, calculating for each node we reach as we go two sets. For node a, we will denote these sets R_a and S_a. The first set is computed afresh for a node each time we reach it, and is meant to specify the set of forward distances from the node at which answer tuples lie. The second set is the union of all values that the first set for the node has received whenever the node has been reached. It is initially null for each node, and is updated each time a node is reached. It is used to prevent duplication of effort, in a manner to be described shortly.

On the first iteration, for each node a reachable from some driver d via some arc, we compute R_a as the union, over all the driver tuple predecessors of a, of the occurrence set for the driver, decrementing the values of the set by 1 to record the traversal of the arc. Letting E denote the edge set for G_A, the computation for the first iteration is

$$R_a = \bigcup_{\substack{(d,a)\epsilon E \\ d \text{ a driver}}} O_d - 1$$

where $O_d - 1$ signifies that each element has its value decreased by one. If $0 \epsilon R_a$, then a is an answer, so pack it off to answer-land, and discard 0 from R_a. We then update R_a and S_a as follows

$$\begin{aligned} R_a &\leftarrow R_a - S_a \\ S_a &\leftarrow S_a \cup R_a \end{aligned}$$

The first time a is encountered, $S_a = \emptyset$, so the effect is simply $S_a \leftarrow R_a$. For the next iteration, traverse out from all nodes a just reached if R_a is not null. For each node b reachable from such a node, compute R_b from the

R sets of its predecessors, in the same way that the R sets were computed from the occurrence sets on the first iteration.

$$R_b \leftarrow \bigcup_{\substack{(a,b)\epsilon E \\ a \text{ reached at} \\ \text{last iteration}}} R_a - 1$$

Again, if $0 \epsilon R_b$, then b is an answer, and should be processed as above. In any event, the R_b and S_b sets should be updated as before. The iteration continues in this fashion until all the updated R sets for the tuples reached during the last iteration are empty.

Intuitively, the R set for a node records the arc radii from the node at which answer tuples lie, as determined from its predecessors in the previous iteration. The propagation of R set values across arcs should need no justification. The subtraction of the current S set value before continuing the propagation is justified by noting that if a value i is in S, then at some previous iteration, i was computed to be an R set value for the node, and the walk to those answers i arcs distant was initiated at that iteration.

For the finite parts of the sets the operations are all straightforward, but the infinite part requires slightly different treatment. If we assume that the driver set cycle is initiated at the lth set and that the occurrence sets contain the elements of Z_p that when added to l, identify the driver sets of the cycle in which the tuple appears, then until the lth iteration the infinite part of the S sets should be left null. The infinite parts of the R sets should be propagated, without decrementing the values of the elements (the decrement is actually implicitly applied to the initial segment, l). At the lth iteration, at which time all finite parts must be empty, the set operations are continued with the infinite parts alone. At this time, the infinite part of the S sets for all tuples reached should be allowed to receive the infinite part of the corresponding R set during the update to S. The only differences in the operations are that 0 is no longer discarded from the R sets and the decrementing is now done $\pmod p$ so $0 - 1 = p - 1$, a value that should be propagated. The set difference operation is performed as for the finite part.

The correctness of this algorithm can be rigorously established by a *reductio ad absurdum* argument showing that if a is one of the answers, then the shortest path from one of the drivers to a, whose length is also in the driver's occurrence set, must be traversed in the walk and will be used to identify a as an answer.

In this alternative to the Henschen and Naqvi algorithm, the redundancy of recording nodes in separate trees and the potential for marking nodes dead perhaps several times over is traded for a proliferation of set operations. Surely, example graphs can be constructed that favor one or the other method. If the graphs are highly connected with many cycles and the values of l and p small enough that a bit vector representation of the finite and infinite parts of the sets can be employed, it seems likely that this alternative would fare well.

We have reviewed the Henschen-Naqvi technique for the simplest case and noted an error in the termination condition whose correction may involve some expensive set comparison operations. Of course, if one is certain that no cycles exist in the G_A graph, or that the G_B graph cycles on $\emptyset$, their method is adequate.The findings of [BANC86] indicate that even with this addition, their method is likely to remain competitive with alternatives. We have also offered a variation on their algorithm that may be more efficient for some data base instances. In presenting the Henschen-Naqvi technique we employed some different representations of the computation, and it is our hope that this description will render their idea accessible to a larger audience.

References

[BANC86] Bancilhon, F., and Ramakrishnan, R., "An Amateur's Introduction to Recursive Query Processing", *Proceedings of SIGMOD '86 International Conference on Management of Data*, pps. 16-52.

[LLOY84] Lloyd, J., *Foundations of Logic Programming*, Springer-Verlag, 1984.

[HENS84] Henschen, L., and Naqvi, S., "On Compiling Queries on Recursive First-Order Data Bases", *JACM*, Vol. 31, January 1984, pps. 47–85.

[REIT78] Reiter, R., "Deductive Question Answering on Relational Data Bases", in *Logic and Databases*, Gallaire, H., and Minker, J, eds., Plenum Press, New York, 1978, pps. 149-177.

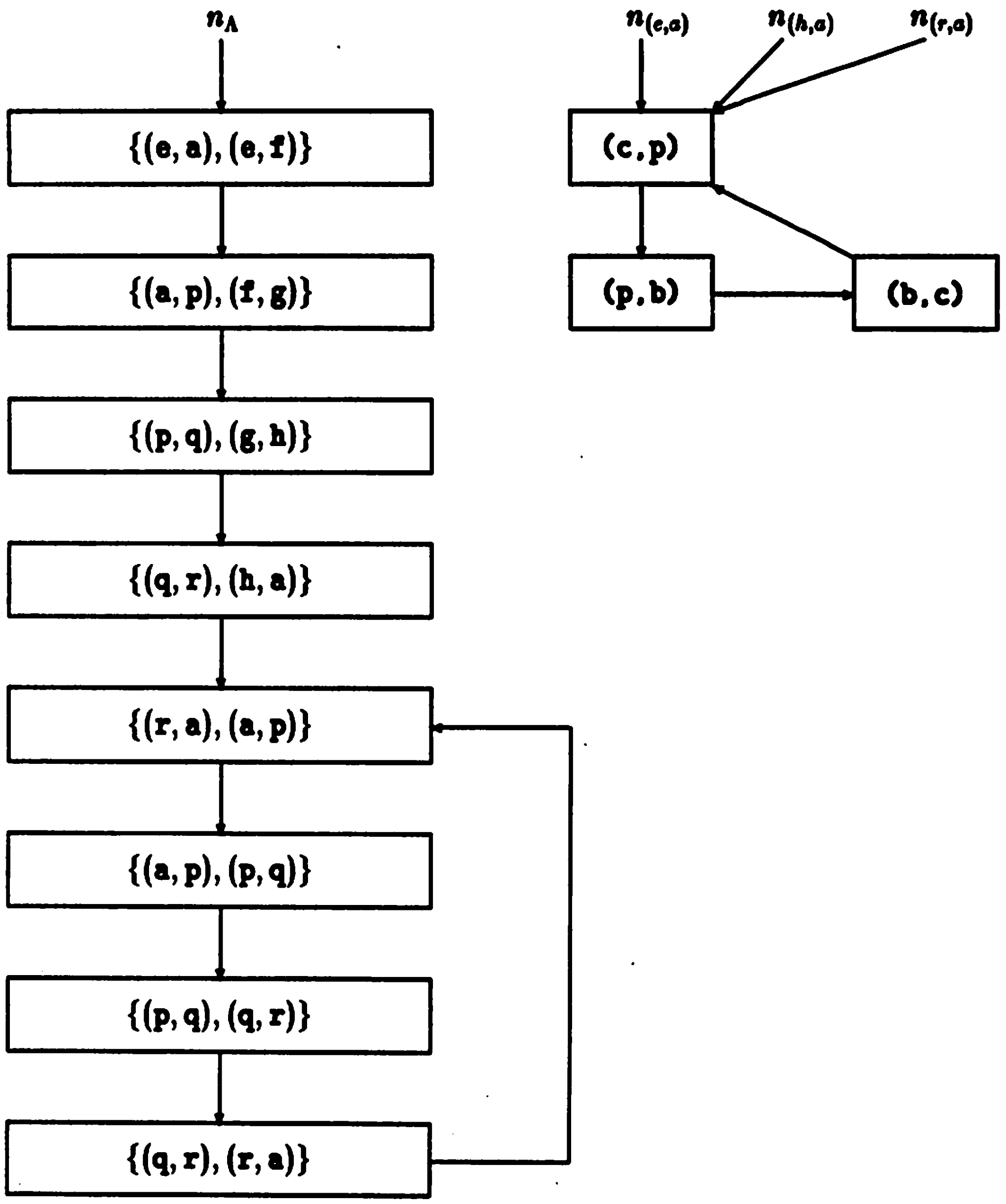

Figure 3: Relevant parts of the graphs G_B and G_A for the first example