AN EXTENDED DISJUNCTIVE NORMAL FORM APPROACH FOR OPTIMIZING RECURSIVE LOGIC QUERIES IN LOOSELY COUPLED ENVIRONMENTS

Kyu-Young Whang
Shamkant B. Navathe[1]

IBM Thomas J. Watson Research Center
P. O. Box 704, Yorktown Heights, New York 10598

ABSTRACT

We present an approach to processing logic queries in loosely coupled environments. We emphasize the importance of the loose coupling technique as a practical solution to provide deductive capabilities to existing DBMSs—especially when an efficient access to a very large database is required in the process of inferencing. We propose the Extended Disjunctive Normal Form (EDNF) as the basis of our approach. The EDNF is an extension of the disjunctive normal form of relational algebra expressions so as to include recursion. The EDNF is well suited for a loosely coupled environment, where an existing DBMS and optimization can be fully exploited. It also serves as a clear, graphical characterization of various recursions that can occur in logic queries. We first present the basic form of the EDNF and then use it as a building block to process a more general class of queries. We extend valid usage of Clark's negation-as-failure evaluation technique to incorporate negation for most practical situations. We also propose new criteria for safety and termination in the presence of negation. To the extent of the authors' knowledge, optimization in loosely coupled environments has not been seriously addressed in previous research. We believe our technique provides significant progress in this direction.

1.0 Introduction

Recently, a number of studies [Ull85, Ban86, Ban86a, Vie86, Boc86, Kri86, Kif86, Kif86a, Loz85, Sac86, Agr87, Jag87a, Mac81, Van86] have concentrated on providing inferencing capabilities to traditional databases. These facilities are geared so that complex views, especially those involving recursion, can be supported. The view mechanisms in the present DBMSs support no derivation of information besides straightforward relational operations. The work to-date in this area has focused on PROLOG as an inference language for DBMSs due to the "natural fit" between PROLOG and the relational data model. Allowing a language such as PROLOG as the query language provides the system with the power of Horn-clause logic as well as the inherent theorem-proving capability.

[1] Permanent Address: Database Systems R. and D. Center, University of Florida, E-460 CSE Bldg., Gainesville, FL 32611.

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.

We propose in this paper an optimization technique of providing these capabilities to an existing DBMS in a loosely coupled manner, especially when an efficient access to a very large database (that cannot fit in main memory) is required in the process of inferencing. Although the loose coupling idea has been introduced in the literature, to the extent of the authors' knowledge, optimization aspects have not been seriously addressed.

1.1 Issues in Recursive Query Processing and Previous Research

In this section we highlight the issues that have been addressed in the previous research. An excellent survey of the work as of mid-1985 appears in [Ban86]. From an overall analysis of the previous research, we conclude that a viable and efficient strategy for processing recursive queries in a deductive database should provide the following characteristics:

- **Feasibility and correctness:** It should have a guaranteed termination and produce a correct result.
- **Coupling efficiency:** For databases having a large amount of data on secondary storage, it should provide an efficient access to the data.
- **Searching efficiency:** It should have the FRD property (*Focus on Relevant Data*) [Vie86, Nic86]. This property may be further divided into FRD-A and FRD-B properties as follows:
  - FRD-A: It should not process irrelevant tuples, which are not necessary to formulate the results.
  - FRD-B: It should not process the relevant tuples repeatedly (i.e., no duplication).

Feasibility and correctness

To determine the feasibility of a given approach, the following issues must be carefully considered.

1. **Cyclic Data:** Cycles in data cause certain evaluation algorithms to get into an infinite loop. For example, suppose we have a relation FLIGHT that shows the origin and destination cities of flights: the relation contains tuples <new york, chicago>, <chicago, dallas>, and <dallas, new york>. Consider the rules:

   reach(X,Y):- flight(X,Y)
   reach(X,Y):- reach(X,Z), flight(Z,Y)

   If a query such as ?reach(X, new york) or ?reach(chicago, Y) is issued against the relation FLIGHT, in some approaches, the

evaluation algorithm gets into infinite recursion. For example, PROLOG has this problem, and so does Counting [Ban86a]. The approach we propose is able to deal with cycles in data and poses no termination problem. In general, approaches classified as "bottom up" [Ban86] are capable of dealing with cycles in data without any difficulty.

2. **Negation:** A number of existing approaches totally disallow negation since it violates the definition of Horn-clause logic. In Section 4 we extend Clark's *negation-as-failure* evaluation technique [CLA78] to handle negation in most practical cases. Extension of negation as failure for general first-order logic databases is discussed in [Naq86]. More scrutiny on negation in conjunction with the closed-world assumption [Rei78] appears in [Naq86a].

3. **Safety:** The safety issue deals with the size of the final or intermediate results of a query. A *query* is *safe* if the final result is finite. We also define *query execution* to be *safe* if all the intermediate results are finite. To guarantee safety of the query and query execution, we require rules to be "bottom-up evaluable" [Ban86]. To handle the case of negation properly, we extend the definition of bottom-up evaluability in Section 4.

4. **Nonlinear recursion:** A recursive rule P :- P1, P2,..., Pn is *linear* if there exists one and only one Pi in the body of the rule that is mutually recursive with P [Ban86]. A rule is *nonlinear* if there is more than one Pi that is mutually recursive with P. Further, a set of rules is linear/nonlinear if every/any rule in that set is linear/nonlinear. The approach we propose is able to handle both linear and nonlinear rules.

Coupling efficiency

A wide range of approaches to providing the DBMS with deductive capabilities have been proposed [Cha85,Vie86,Boc86a,Ban86,Kif86]. In one set of approaches, classified as *tight coupling*, a DBMS is extended to incorporate rule management and inferencing, thereby integrating the database capabilities with deductive capabilities. However, such approaches have not exploited the query optimization techniques existing in the DBMSs.

The other set of approaches is classified as *loose coupling*. In the loose coupling philosophy, a DBMS is considered a complete, independent system. The communication with the DBMS is supposed to occur at the level of a database query language (in our case SQL). This approach has the following potential advantages:

- It allows one to use a relational DBMS without having to redesign (and reimplement) it.
- It allows the full power of relational query optimization to be exploited while retaining the option of performing additional optimization in the logic program itself.

In loose coupling, however, care must be taken to achieve efficient database access. For example, in a technique that we term *naive loose coupling*, requests are made to the DBMS whenever the necessary data reside in the DBMS. However, this technique may cause excessive database access that could lead to tuple-by-tuple access to data in the worst case. Typically, interpretation (vs. compilation) is dominant in naive coupling.

In the type of loose coupling we propose (*smart loose coupling*), compilation [Ull85, Hen84.] is preferred to interpretation. Thus, a logic query is compiled into a small number of database queries (with possible iterative constructs) to run on the DBMS. Since queries are formulated at the granularity of entire relations or collections of them, any tuple-by-tuple transfer of data between the DBMS and the logic programming environment is strictly avoided. For example, suppose we have a rule a(X,Y) :- b(X,Z), c(Z,Y), where there are 1000 tuples each satisfying the predicates b and c. Consider a query ?a(X,Y). In naive loose coupling, with a Prolog-like depth-first search strategy, processing the query requires 1001 calls to the DBMS. In smart loose coupling, on the othe hand, we need only one database query, which is the join of b and c.

The technique used in PROSQL [Cha85] allows both naive and smart loose coupling. However, in this system, the user is reponsible for the translation between the logic program and SQL queries. Jarke et al. [Jar84] discusse a loose coupling approach, but it is limited to a nonrecursive part of the Prolog program without negation. The system EDUCE/DEDGIN [Boc86,Nic86,Vie86] supports both tight and loose couplings. For the part of loose coupling, the system treats PROLOG as a host language for general application development and poses requests for data to INGRES DBMS whenever necessary (i.e., when the data reside in the DBMS) while processing with typical PROLOG interpreter. For the part of tight coupling, the deductive component called DEDGIN [Vie86] looks at the function-free Horn-clause subset of PROLOG assertions and couples tightly to the same DBMS by directly calling the internal access methods.

We note that not much work has been done on coupling efficiency, except in [Kri86] and [Cer86]. Krishnamurthy and Zaniolo [Kri86] briefly discuss cost equations that can be used in loose coupling. Ceri, Gottlob, and Wiederhold [Cer86] assumes that all the query processing is done in main memory with a memory-resident copy of data and proposes an algorithm to load the data from the DBMS to main memory intelligently. In this approach, however, the query processing and optimization capabilities of the DBMS are not utilized. We believe coupling efficiency is an important issue to be addressed. Our approach specifically deals with this problem—especially, in a loosely coupled environment.

Searching efficiency

We have proposed two categories in the FRD property: FRD-A and FRD-B. Many papers address searching efficiency. For example, Sideways Capture Rules [Ull85], Magic Sets [Ban86a], Counting [Ban86a, Sac86], Filtering [Kif86,Kif86a], etc. dwell largely on the FRD-A property. The Semi-Naive evaluation technique [Ban85] addresses the FRD-B property in the case of linear rules. However, the technique requires the use of relational algebra expressions to calculate differentials explicitly, and sometimes these expressions are too complicated to obtain. A *differential* is an incremental result from each iteration during evaluation. Our approach provides a simple efficient technique of achieving the FRD-B property by implicitly (i.e., without using a formula) calculating the differentials (See Section 3.2). This technique is applicable to any set of linear rules. We do not cover the FRD-A property in this paper, but we believe that it can be superimposed by adding an additional rule modification phase.

Another technique of achieving searching efficiency is based on extended relational algebra [Aho79,Agr87,Dev86]: pushing the se-

lection operator across the fixed point operator to the base relations as close as possible. This technique is a heuristic optimization at the level of a relational algebra.

1.2 Our Approach

We consider the class of logic queries that are expressed in function-free Horn-clause Logic with extension to incorporate negation. For the safety of query computation [Ban86, Kri86], we further restrict the rules to be "bottom-up evaluable" [Ban86] with a modified definition to accommodate negation (see Section 4).

In a nutshell, our approach is to decompose a logic query composed from a set of rules into units termed *Extended Disjunctive Normal Form (EDNF)* components. The term EDNF is derived from disjunctive normal form (more about this later in Section 3) that can be applied to Boolean expressions. In the context of relational algebra, conjunctions refer to joins and disjunctions to unions. The EDNF is an extension that allows us to deal with recursion in addition to relational operations.

We show that the EDNF is well suited for a loosely coupled environment and discuss how it reduces the call to the DBMS, which is a costly operation in such an environment. We show that a logic query can be decomposed and transformed into an equivalent forest of EDNF trees. We also present an algorithm to process the query using the EDNF trees. Using a fixed point formalism, we show that this transformation coupled with the processing algorithm is sound in that it generates the correct result. We discuss interesting ramifications of the EDNF formalism including graphical characterization of the complexity of recursion and corresponding efficient query processing algorithms. Finally, we handle negation in most practical situations by extending valid usage of Clark's negation as failure. We propose new criteria for safety and termination in the presence of negation.

Currently, we are implementing an inference engine based on the EDNF approach for an expert system shell, SQL Inference Engine, using an interface language derived from a version of SYLLOG [Wal83]. The system utilizes data stored in the underlying DBMS: SQL/DS.

The rest of the paper is organized as follows. In Section 2 we provide the motivation to our approach and define the notation. In Section 3 we present the concept of the EDNF and prove that the evaluation based on the EDNF is correct. In Section 4 we extend the EDNF to include negation and present the technique of processing logic queries by decomposing them into EDNF components. We discuss advantages of the EDNF approach in Section 5 and summarize our results in Section 6.

2.0 Motivation Behind Our Approach

2.1 Background

We start with the rule/goal graph of Ullman [Ull85] to describe the data structure to represent a logic query. The rule/goal graph represents a set of rules by creating *rule nodes*—one for each rule with a specific adornment [Ban86a] for the variables and *goal nodes*—one for each predicate with a specific adornment for the arguments. An *adornment* is a string of b's and 'f's indicating the binding status of the variables in a rule or the arguments in a goal. The symbol 'b' means that the corresponding variable or argument is bound, i.e., instantiated, whereas the symbol 'f' means that it is free, i.e., uninstantiated. Thus, if a node r has k variables, there are 2^k nodes marked r^a where a is the adornment. For example, r^{bff} shows an instance of rule r with the first variable bound and the other two free. Nodes are similarly created for all possible bindings of arguments of a predicate. To describe our scheme, we need to modify the rule/goal graph slightly. We call the modified one *the query graph*. While the rule/goal graph is composed for the *entire set of rules*, we construct a query graph for a *specific given query*. In addition, the query fixes a particular value (we call it the *binding value*) for a bound argument. In essence, by using the query graph, we trim the rule/goal graph by retaining only those nodes relevant to a specific query. The method of connecting the nodes by arcs remains the same as in the rule/goal graph, except that we distinguish different binding values. Note that generating rule/goal graphs to account for all possible values would not be practically feasible because the number of distinct values is potentially infinite. To illustrate the query graph, consider the following rules:

r1: a(X,Y):- b(X,Z), c(Z,Y)

r2: b(X,Z):- e(X,L), f(L,Z)

r3: e(X,L):- g(X,K), h(K,L)

Consider the query ?a(X,5). It corresponds to the node a^{fb} in the rule/goal graph with the specific binding value of 5 for the second argument. We label this node as a(f/X, b(5)/Y). The completed query graph is shown in Figure 1.

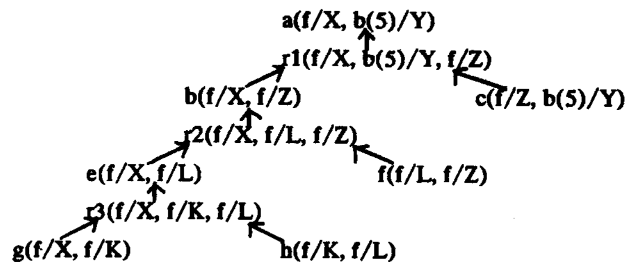

Figure 1. The query graph for the query ?a(X,5).

In constructing the query graph, we enumerate all variables involved in that rule and show appropriate bindings. Binding information is passed down from the query node by means of unification. Note that the variable L in rule 2 was unified with the variable Z in the goal node b(f/X, f/Z). In the graph the rule node (*AND node*) implies the conjunction of goals connected by incoming arcs; the goal node (*OR node*) implies the disjunction of (bodies) of rules connected by incoming arcs [Ull85].

Now, let us construct the query graph of a recursive query. Consider the rules:

r1: a(X,Y) :- c(X,L), e(L,Y)

r2: a(V,W) :- d(M,V), g(V,Z), a(Z,W)

Suppose the question ?a(X,5) is asked. Then, the query graph is as in Figure 2.

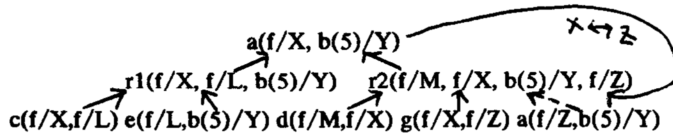

Figure 2. Query graph for the recursive query ?a(X,5).

In Figure 2, we draw an arc from a( ) to r2( ) because the node a(f/*, b(5)/*) appears repetitively with the same binding information. Since we cannot unify the variables/arguments in an existing node (i.e., one already constructed), we use *mapping* of variable names; in this case X in node a( ) is mapped to Z in node r2( ). When we later evaluate the query graph, we have to *resolve* the mapping to establish the correct association of the mapped variable with those in other nodes. A mapping is resolved by replacing the variable name in the node at the tail of the arc with the name at the head side of the arc. (Note that in [Ull85] mapping is implicit in the rule/goal graph, and the substantiation algorithms have to keep track of it.) For example, in Figure 2, a broken arc shows how the node a( ) looks like when the mapping has been resolved. Notice that the mapping does not necessarily indicate presence of recursion (directed cycle) because mapping is also needed when an undirected cycle is formed. An undirected cycle typically results when more than one branch of the graph refers to the same goal node with the same binding information.

2.2 Scope for Improvement

Associated with the rule/goal graph, there are substantiation algorithms that compute the relation for each node according to capture rules [Ull85]. A relation is associated with each node in the rule/goal graph. This relation is a set of tuples that satisfy the constraints implied for the node by the graph. From now on, we shall use a node in the graph and its relation as being synonymous.

In the straightforward application of these substantiation algorithms, we have to create a temporary relation for each node in the query graph, since the query is processed by evaluating each node according to the structure dictated by the rule/goal graph. Here, we observe some potential for improvement:

1. Creating many temporary relations not only takes a potentially excessive amount of storage space but causes an adverse effect on performance in a loosely coupled environment. In particular, a join between a temporary relation in memory and a DBMS relation could cause as many calls to the DBMS as the numbers of tuples in the temporary relation. Therefore, we need to minimize the number of temporary relations created in evaluating a query.
2. The structure of the rule/goal graph is inherited from the user-written rules. Thus, the execution structure (such as the ordering of relations), and accordingly the performance, is heavily dependent on these rules. This counters the principle of data independence. Our aim is to eliminate this limitation by "normalizing" the query to keep only semantic information that is necessary to evaluate the query.
3. The rule/goal graph approach does not take advantage of existing DBMS optimization.

Example 1: Consider the rules and the query in Figure 1. In this case, we need six temporary relations (one for each nonleaf rule or goal node). Besides, the "basic capture rule" in [Ull85] indicates that the joins must be evaluated in the order (((g Join h) Join f) Join c). However, we know that we can process this query with only one temporary relation (for the result) in main memory and that the DBMS optimization can choose any join ordering that provides the best performance. The normalized query is (g Join h Join f Join c). It can be translated to a database query as follows:

```
SELECT   g.1, c.2
FROM     g, h, f, c
WHERE    g.2=h.1 AND h.2=f.1 AND f.2=c.1 AND c.2=5
```

In the above query, projection lists and join conditions are shown using positional identifiers for attributes.

End Example 1 ◊

3.0 The Extended Disjunctive Normal Form

In this section we define the concept of an Extended Disjunctive Normal Form (EDNF) of a query graph. The Extended Disjunctive Normal Form is an extension of the disjunctive normal form of relational algebra expressions so as to include recursion.

The purpose of our technique is to process a query in such a way as to avoid the shortcomings of the straightforward evaluation of the rule/goal graph that are discussed in the last section. In Section 3.1 we present the definition of the EDNF and we discuss the algorithm to transform the query graph into the EDNF representation. In Section 3.2 we present an algorithm to generate the answer to the query using the EDNF. In Section 3.3, using a fixed point formalism, we prove that the answer obtained from the EDNF is indeed the answer to the original query. The EDNF can be constructed only for those query graphs in which the query goal (defined in Section 3.1) is included in any directed cycle. The application of the EDNF to more general queries (i.e., when some cycles do not go through the query goal) is discussed in Section 4.

3.1 Definition of the EDNF

For a given set of rules, a query graph is constructed to represent a specific query against this set. We call the root of such a graph the *query goal.* Thus, the relation corresponding to the query goal is the answer to the query. The EDNF representation of a query graph has the following characteristics:

1. It is a set of two-level trees.
2. The root of each such tree is the query goal.
3. Each tree has one or more leaf nodes that are base relations (i.e., they are not temporary relations). A leaf node in the tree must be a leaf node in the query graph.
4. A tree may have one or more loops on the root indicating recursion. We call such a tree a *looped tree.* If a looped tree has a single loop, we call it a *single-looped tree*; otherwise, we call it a *multilooped tree.* The set of EDNF trees is called an *EDNF Forest.*
5. There is only one temporary relation, which is the root (query goal) of all the trees.

The EDNF minimizes the number of temporary relations since it has only one temporary relation, which is essential for storing the result of the query. It also normalizes the query into a two-level flat structure, eliminating the arbitrary structure imposed by user-written rules. Figure 3 shows the EDNF of the query graph in Figure 1.

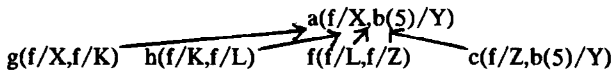

Figure 3. The EDNF for the nonrecursive query in Figure 1.

Figure 4 shows the EDNF of the recursive query graph in Figure 2.

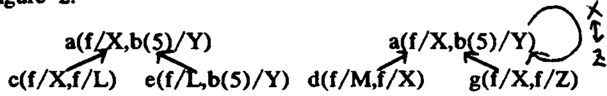

Figure 4. The EDNF of the recursive query in Figure 2.

The loop on the root node in Figure 4 actually stands for the root node used as a leaf node in its own definition. The mapping in the loop represents that the variable name X is to be replaced with the variable name Z when resolved as defined in Section 2.1. Thus, the second tree (in the *folded form*) in Figure 4 can be alternately shown as in Figure 5. We call it the *unfolded form* of the looped tree. In the unfolded form, the loops appear without mapping just to indicate the presence of recursion. If the tree has multiple loops, the root appears as a leaf multiple times—once for each loop.

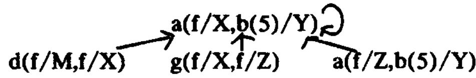

Figure 5. The unfolded form of the looped tree in Figure 4.

Generation of an EDNF forest

We present a description of an intuitive (top-down) procedure for converting the query graph into the corresponding EDNF. A formal description of the algorithm based on bottom-up construction is in [Wha87].

1. The query graph is traversed by following one of the alternative branches at every OR node (a nonleaf goal node). For an AND node (a rule node), if it has a leaf node as a child, the child is attached to the output tree as a leaf node. Otherwise, all branches coming up into that AND node are traversed. When all these AND branches reach leaf nodes, one EDNF tree has been constructed.
2. After the first visit, whenever the query goal is visited again, we treat it as a leaf node. Since this indicates recursion, we mark the tree as a looped tree. At some point in the path, if multiple cycles bifurcate from an AND node, it forms a multi-looped tree. On the other hand, if multiple cycles bifurcate from an OR node, we obtain multiple single-looped trees rather than one multilooped tree. This results in an interesting graphical characterization of linear or nonlinear rules. The presence of any *multilooped tree* in the EDNF indicates that the set of rules used in the query graph is *nonlinear*; if there are only *nonlooped* and *single-looped trees*, then the set of rules is *linear*.
3. Whenever a leaf node is encountered, we record on the arc the mapping accumulated (using function composition) through the entire path from the root to the leaf unless it is the identity mapping. Details of manipulation of the mapping are described in [Wha87].

For ease of understanding, the algorithm as described here produces looped trees in the folded form. To actually process the query, however, we need to transform the trees to the unfolded form. The algorithm in [Wha87] directly produces the looped trees in the unfolded form.

Example 2: Figure 6 and Figure 7 show query graphs containing nonlinear and linear recursive rules, respectively, and their EDNFs. For simplicity, we represent nodes simply as OR and AND. Only the root and leaf nodes are shown with predicate names.

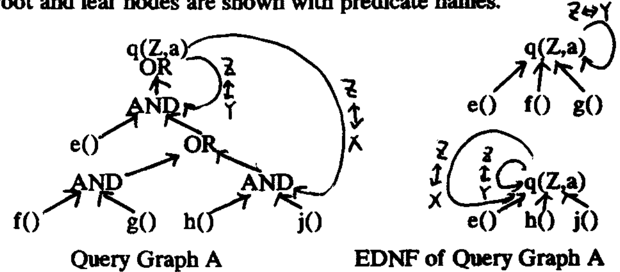

Figure 6. A query graph and the EDNF involving nonlinear rules.

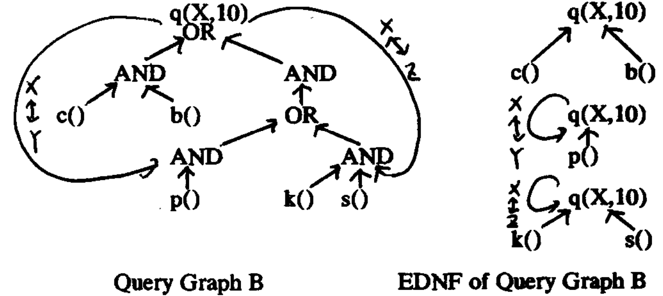

Figure 7. A Query Graph and the EDNF involving only linear rules.

Note that in Figure 6 the bifurcation of the two cycles occurs at an AND node (thus generating a double loop), whereas in Figure 7 it occurs at an OR node (thus generating single loops).

End Example 2 ◊

In constructing the EDNF, some care must be taken not to lose any binding information propagating *upwards* through the query graph. These bindings may come from the rule heads. Two typical cases are shown below:

Case 1:	a(X,Y,0)	:- ....
Case 2:	b(X,X)	:- ....

Since in the EDNF, all the nonleaf nodes are eliminated, these bindings will be lost. We solve this problem by modifying the above rules *before* conversion into EDNF as follows:

Case 1:	a(X,Y,Z)	:- ...., (Z=0)
Case 2:	b(X,Y)	:- ...., (X=Y)

Thus, basically we introduce new variables in the heads and move the binding information into the body of the rules. The transformation we use is consistent with the *general form* of the clause in [Cla78].[2]

[2] This is brought up again in the discussion of negation in Section 4.

3.2 Processing Queries Using the EDNF

The EDNF is a data structure representing a query. We now present an algorithm to process a query using this data structure. Let us use the following notation:

NT: the set of nonlooped trees in the EDNF
LT: the set of looped trees in the EDNF

The algorithm is described below:

```
Algorithm PROCESS__QUERY (NT, LT, Result)
    1. Result:= φ
    2. For each tree t ∈ NT
           Result:= Result ∪ EVAL(t)
       Endfor
    3. repeat until no change in result
           For each tree t' ∈ LT,
               Result:= Result ∪ EVAL(t')
           Endfor
End PROCESS__QUERY
```

The algorithm EVAL simply evaluates a nonrecursive query represented by a nonlooped tree. When evaluating a looped tree, it disregards the loops from the unfolded form of the looped tree (see Figure 5). A nonlooped tree is evaluated as a join of all the leaf nodes in the tree projecting the result for the arguments in the root (query goal). Such a join request is submitted to the DBMS. Join conditions are derived from the matching variables in different nodes. If the leaf is an evaluable (arithmetic) predicate such as X > Y, then it is treated as a condition. If all the variables in a condition come from database relations, the condition is imbedded in the database query. If there is a safety dependency [Zan86] among variables in the evaluable predicates, these predicates are evaluated based on the order of the dependencies. A set of variables Y is *safety dependent* on a set of variables X ($X \rightarrow Y$) if there is a finite number of Y values once values of X are fixed.

Example 3: In Figure 8 there is a safety dependency $X \rightarrow Y \rightarrow Z$. Hence, the evaluation of the tree proceeds as follows. First, $\Pi_{1,4} \infty(p,s,\text{cond}(p.2 = s.1))$ is processed by issuing a query to the DBMS.[3] Next, for each tuple in the result, we

1. evaluate Y=2*X and bind Y;
2. evaluate Z=Y+3 and bind Z;
3. check the condition Z>W to select tuples from the result of join.

Note that both Z and W are bound by the time the condition is checked. Otherwise, q(X,Y,Z) would not be "bottom-up evaluable."

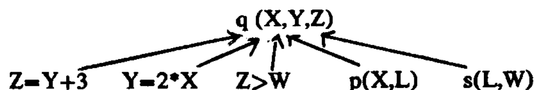

Figure 8. A general nonlooped tree.

End Example 3 ◊

It is interesting to note that the algorithm EVAL is similar to the algorithm needed to process a domain-calculus query as in QBE [Zlo77]. We can therefore take advantage of optimization techniques developed for this class of DBMS queries (e.g., see [Wha85]).

A salient feature of the algorithm PROCESS__QUERY is that nonlooped trees are evaluated and unioned "only once." The iterative procedure is applied only to the looped trees. This makes a significant improvement over what is implicitly indicated in Tarski's [Tar55] formalism. There, nonlooped trees would be unioned again and again at each iteration. We believe that some of the existing naive approaches do exhibit this drawback. A natural consequence of the EDNF approach is to isolate the portion of the query to be processed iteratively to a minimal possible scope.

Algorithm PROCESS__QUERY is the basic algorithm to process looped trees. We can construct alternative processing algorithms suitable for processing looped trees in different situations as in the following examples.

Example 4:

We describe an algorithm that can process the queries derived from linear rules efficiently. These queries contain only nonlooped and single-looped trees. This algorithm avoids duplicate processing of relevant data, i.e., it satisfies the FRD-B property. Further, it does not require explicit calculation of differentials; instead, it obtains them implicitly.

For the sake of simplicity, we describe the algorithm using a specific example shown in Figure 4 and Figure 5. First, algorithm PROCESS__QUERY described above evaluates the query as in Figure 9.

1) $a_0 := \Pi_{1,4} \infty(c,e,\text{cond}(c.2 = e.1))$

2) repeat (incrementing i) until $a_i = a_{i-1}$

$$a_i := a_{i-1} \cup \Pi_{4,2} \infty(a_{i-1}, d, g, \text{cond}(a_{i-1}.1 = g.2 \wedge d.2 = g.1))$$

Figure 9. Iterative evaluation of the EDNF in Figure 4.

Note that a straightforward application of this procedure involves some redundant computation. The result of Step 2 at iteration $i-1$, a_{i-1}, is processed (i.e., joined with d and e) at iteration i to produce a_i. However, this processing is duplicated at iterations $i+k$, where $k = 1, 2, 3$, etc., because a_{i-1} is a subset of all a_{i+k}'s.

Algorithm ONE-PASS described below completely avoids this redundancy. In this example we consider a situation in which the result (relation a) is constructed in main memory, and relations c, e, d, and g are in the DBMS.

Algorithm ONE-PASS

1) compute($c(X,L)$&$e(L,5)$&
check-unique-and-addnewtuple($a(X,5)$))

2) compute($a(Z,5)$&$d(M,X)$&$g(X,Z)$&
check-unique-and-addnewtuple-at-the-bottom($a(X,5)$))

In algorithm ONE-PASS, the predicate compute finds all variable bindings that satisfy the goal passed as the argument. Step 1 proc-

[3] Here and throughout the paper, the symbol ∞ represents the prefix join operator, and Π the projection operator.

esses the nonlooped tree and constructs the initial value of relation a (i.e., a_0), and Step 2 processes the looped tree. Step 1 and Step 2 correspond to Step 1 and Step 2 in Figure 9, respectively. For convenience, we described the algorithm in a Prolog-like fashion. In actuality, however, the algorithm generates calls to the DBMS in as big queries as possible to access the data in the DBMS. In this example the following database queries will be composed: $\Pi_{1,4} \infty(c,e, \text{cond}(c.2 = e.1))$ and $\Pi_{2,4} \infty(d,g, \text{cond}(d.2 = g.1 \wedge g.2 = z))$, where z is a specific value of Z bound by a tuple of a. The algorithm is further illustrated in Figure 10.

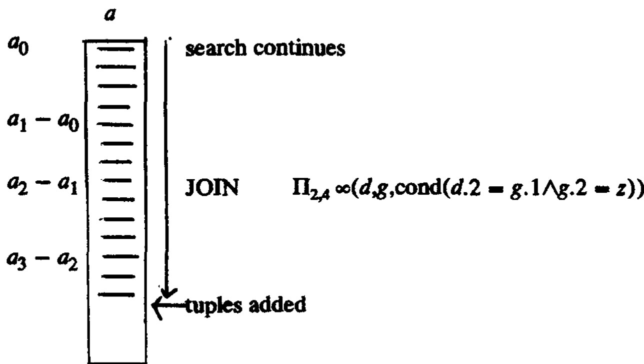

Figure 10. The ONE-PASS algorithm that avoids duplicate processing of relevant data.

Step 2 of algorithm ONE-PASS does not contain explicit iteration. Instead, it makes a "single pass" through relation a, while adding new tuples from partial results to the bottom of a at the same time. New tuples added are checked for uniqueness before insertion. (Note that set union in Figure 9 should also check for uniqueness. This applies to any iterative evaluation techniques.) When no more new tuples are added, the computation stops.

In effect, the algorithm adds $(a_{i+1} - a_i)$ as a result of processing $(a_i - a_{i-1})$ (i.e., joining $(a_i - a_{i-1})$ with $\infty(d,g)$), where i is a specific iteration in Figure 9. That is, it obtains a differential at iteration $i+1$ by processing only the differential obtained at iteration i. Thus, it avoids duplicate processing of tuples in a_{i-1} when producing $(a_{i+1} - a_i)$. Note that the algorithm does not require an explicit relational algebra expression to calculate the differential. A related work based on differentials has been presented in [Cai87] in the context of programming languages.

The one-pass technique is possible for two reasons:

1. Only one relation (a in this case) is evaluated as the result: the EDNF representation has this property since it requires only one temporary relation that corresponds to the query goal.
2. Only single-looped trees are considered: only linear sets of rules provide this property.

End Example 4 ◊

Example 5:

Although the class of queries that can be represented in the EDNF is far larger than that of the simple transitive closure, their EDNF representations resemble each other. A transitive closure appears in the EDNF as one nonlooped tree and one single-looped tree, each having one leaf. Hence, any efficient techniques to process the transitive closure [Ioa86, Yu87] can be applied to the EDNF with little modification.

End Example 5 ◊

Transitive closure vs. linear rules

Similarity between transitive closure and linear rules has been demonstrated in [Jag87] for the case of a single recursive rule (with a nonrecursive exit rule). Our scheme shows the same observation can be made for a *set* of linear rules. We simply treat a set of nonlooped trees effectively as one nonlooped tree. The leaf of the new tree is the union (i.e., disjunction) of the sets of leaves from the orignial trees. Similarly, we treat a set of single-looped trees as one single-looped tree whose leaf is the union of the sets of leaves from the original single-looped trees. Thus, the function for which the transitive closure is defined is the union of the joins of relations rather than a single relation. Our scheme also provides interesting insights into nonlinear effects. Although nonlinear rules cannot be precisely represented as a transitive closure, any nonlinear effect is cleanly isolated in the form of multilooped trees.

3.3 Least Fixed Point Formalization of the EDNF

Our goal in this section is to use the fixed point formalism to show that our proposed transformation of the query graph into the EDNF obtains the same result as that of the original query.

Theorem 1: Algorithm PROCESS_QUERY produces the correct result to the query.

To prove Theorem 1, we need to introduce several theorems and lemmas regarding fixed points. Detailed proofs of these theorems and lemmas appear in [Wha87].

Theorem 2 [Tarski]:

Given:

A *poset* $(S, \leq, o)$ is a partially ordered set with respect to a binary relation $\leq$. S has a unique minimum element o. Also, there cannot be an infinite sequence of strictly ascending elements of S. We call this condition the *Ascending Chain Condition (ACC)*. $f: S \to S$ is a monotone increasing function with respect to $\leq$. A *fixed point* of f is defined as an element x of the domain of f such that $f(x) = x$. The *least fixed point* of f, $(LFP(f))$, is the smallest fixed point with respect to $\leq$.

Then:

$LFP(f) = f^k(o)$ for some finite k, i.e., the LFP can be computed as the value x at the termination of the following program:

```
x: = o;
while f(x) > x
        x: = f(x);
end;
```

Theorem 2 can be extended in a straightforward manner to a system of equations. Thus, we define the fixed point of a vector of functions $\bar{F}$ with the equation:

$$\bar{X} = \bar{F}(\bar{X}), \quad [1]$$

where $\bar{X}$ is a vector of fixed point variables $\langle X_1, X_2, \ldots, X_n \rangle$. Equivalently, we have

$$X_i = f_i(X_1, \ldots \cdots, X_n), i = 1 \text{ to } n \quad [2]$$

Lemma 1: Inheritance of Ascending Chain Condition

Given posets $(S_i, \leq_i, o_i)$, i=1,2, ...,n, each with the ACC. Then, $S = (\underset{i=1}{X} S_i, \leq, < o_1, o_2, \ldots >)$ is also a poset with the ACC. The binary relation $\leq$ is defined as:

$$< X_1, X_2, \ldots, X_n > \leq < Y_1, Y_2, \ldots, Y_n > \text{ iff } X_i \leq_i Y_i \text{ for all i.}$$

Lemma 2: Inheritance of Monotonicity

Let $(f_i: S \rightarrow S_i)$ be monotone increasing; i.e., if $\bar{X} \leq \bar{Y}$, then $f_i(\bar{X}) \leq_i f_i(\bar{Y})$. Then,
$\bar{F}: S \rightarrow S \ni \bar{F}(X_1, \ldots, X_n) = < f_1(X_1, \ldots, X_n), \ldots, f_n(X_1, \ldots, X_n) >$
is also monotone increasing.

Theorem 3: Given S in Lemma 1 and $\bar{F}$ in Lemma 2, LFP($\bar{F}$) can be computed as follows:

$\bar{X} := \bar{o}$;
while $F(\bar{X}) > \bar{X}$
$\quad\quad \bar{X} := F(\bar{X})$;
end;

Theorem 4 [Kildall]: Given :S in Lemma 1 and $\bar{F}$ in Lemma 2, LFP($\bar{F}$) can be computed as follows:

$< X_1, X_2, \ldots, X_n > := < o_1, o_2, \ldots, o_n >$;
while $\exists i \in \{1, \ldots, n\} \ni (X_i < f_i(X_1, X_2, \ldots, X_n))$

$$X_i := f_i(X_1, X_2, \ldots, X_n) \quad [3]$$

end;

Theorem 4 [Kil73] basically says that LFP($\bar{F}$) can be computed by iterating on individual equations. We shall refer to one step of this iteration with any one equation as an *elementary execution* . In addition, Theorem 4 allows us to choose any order of performing the elementary executions without affecting the least fixed point.

To prove Theorem 1 we proceed as follows. We subsequently illustrate the proof with an example.

1. First, we model the query graph as a system of equations as in Eqs. [2]. One equation is created for each node in the graph. We call it the *query graph system of equations (QGSE)*. Similarly, we model the EDNF forest as a system of equations, but in a slightly modified form to introduce an explicit OR node. We call it the *modified EDNF system of equations (MEDNFSE).*
2. Next, we define a *macro execution* of the system of equations QGSE as the set of all elementary executions that are performed according to the "partial order" defined by the query graph in its unfolded form with the cycles broken. Note that, from Theorem 4, we have the freedom of choosing any order of elementary executions without affecting the result. Similarly, we define a macro execution for the modified EDNF. From the algorithm constructing the EDNF forest, we conclude that one iteration of the macro execution on the QGSE and one on the MEDNFSE produce the same result for the query goal (i.e., the root node), since the result of the latter is the disjunctive normal form of the result of the former. Therefore, both iterative macro executions converge to the same least fixed point.
3. Finally, we show that the iterative macro execution terminates. Since the relational algebra expressions include only selection, projection, cartesian product or join, and union, they are monotone increasing functions [Aho79]. (We disallow set difference by excluding negation involved in a cycle as discussed in Section 4.1.) Since the "values" for the relations are limited by the cartesian product of all values in the base relations and those derived by evaluable predicates subject to safety conditions (as in Example 3), the ascending chain condition (ACC) is satisfied with respect to set inclusion. Thus, the iteration must terminate.

Example 6: Consider the following rules:

r1: q(X,Y) :- a(X,Z), b(Z,Y)
r2: q(X,Y) :- c(X,Z), d(Z,Y)
r3: d(Z,Y) :- e(Z,Y)
r4: d(Z,Y) :- f(Z,L), q(L,Y)

The query graph for ?q(X,Y) and the corresponding EDNF forest are shown in Figure 11 and Figure 12, respectively.

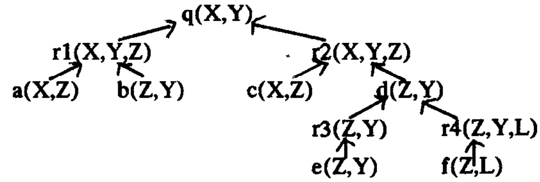

Figure 11. Query graph for ?q(X,Y).

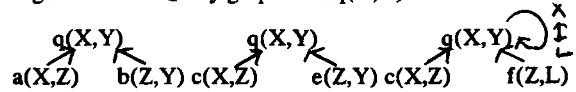

Figure 12. The EDNF forest of the query in Figure 11.

First, we modify the EDNF forest to introduce an explicit OR as in Figure 13 below.

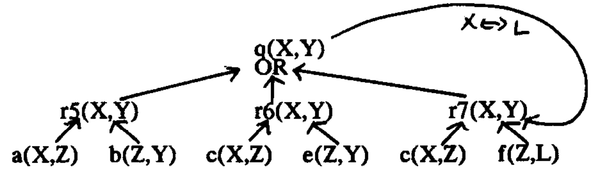

Figure 13. A modification of the EDNF forest in Figure 12.

If we apply the bottom-up capture rule and the substantiation algorithm [Ull85] to the modified graph in Figure 13, it is easy to show that the result remains the same as the result of PROCESS_QUERY for the EDNF in Figure 12. Basically, in Figure 13, the two disjuncts namely, r5 and r6, are *repeatedly* unioned with the query goal q (which is an inefficient technique), while in the EDNF they are unioned *only once*. Now, we shall prove that the query graph in Figure 11 and the modified EDNF in Figure 13 are equivalent in that they produce the same result.

Using relational algebra the query graph above can be represented by the following system of equations. Without loss of generality, we choose one total order of the equations that satisfies the partial order defined by the query graph.

$$r3 = e$$
$$r4 = \Pi_{1,4,2} \infty(f,q,\text{cond}(f.2 = q.1))$$
$$d = \cup (r3, \Pi_{1,2} r4)$$
$$r2 = \Pi_{1,4,2} \infty(c,d, \text{cond}(c.2 = d.1)) \quad \text{[QGSE]}$$
$$r1 = \Pi_{1,4,2} \infty(a,b, \text{cond}(a.2 = b.1))$$
$$q = \cup (\Pi_{1,2} r1, \Pi_{1,2} r2)$$

The modified EDNF in Figure 13 can be represented as follows:

$$r5 = \Pi_{1,4,2} \infty(a,b,\text{cond}(a.2 = b.1))$$
$$r6 = \Pi_{1,4,2} \infty(c,e, \text{cond}(c.2 = e.1))$$
$$r7 = \Pi_{1,6,2,4} \infty(c,f,q,\text{cond}(c.2 = f.1 \wedge f.2 = q.1)) \quad \text{[MEDNFSE]}$$
$$q = \cup (\pi_{1,2} r5, \Pi_{1,2} r6, \Pi_{1,2} r7)$$

We now define a *macro execution* for the QGSE as the set of all elementary executions in the QGSE that are performed according to the partial order defined by the query graph in its unfolded form with the cycle broken. Similarly, we can define a macro execution for the MEDNFSE.

As a result of a macro execution on QGSE, we obtain

$$r3_{n,1} = e$$
$$r4_{n,2} = \Pi_{1,4,2} \infty(f,q_{n-1}, \text{cond}(f.2 = q_{n-1}.1))$$
$$d_{n,3} = \cup (r3_{n,1}, \Pi_{1,2} r4_{n,2})$$
$$r2_{n,4} = \Pi_{1,4,2} \infty(c,d_{n,3}, \text{cond}(c.2 = d_{n,3}.1)) \quad \text{[QGSE]}$$
$$r1_{n,5} = \Pi_{1,4,2} \infty(a,b, \text{cond}(a.2 = b.1))$$
$$q_n = q_{n,6} = \cup (\Pi_{1,2} r1_{n,5}, \Pi_{1,2} r2_{n,4})$$

Here, the first subscript stands for the iteration number of the macro execution, and the second the iteration number of the elementary execution within a macro execution. By substituting $r1_{n,5}$ and $r2_{n,4}$,

$$\begin{aligned} q_n &= \cup (\Pi_{1,4} \infty(a,b), \Pi_{1,4} \infty(c,d_{n,3})) \\ &= \cup (\Pi_{1,4} \infty(a,b), \Pi_{1,4} \infty(c, \cup (e, \Pi_{1,4} \infty(f, q_{n-1})))) \quad \text{[QG-n]} \\ &= \cup (\Pi_{1,4} \infty(a,b), \Pi_{1,4} \infty(c,e), \Pi_{1,6} \infty(c,f, q_{n-1})) \end{aligned}$$

where q_n is the value at the *n-th iteration*. For simplicity, we dropped the join conditions in the above equations. Similarly, the macro execution on MEDNFSE produces

$$q_n = \cup (\Pi_{1,4} \infty(a,b), \Pi_{1,4} \infty(c,e), \Pi_{1,6} \infty(c,f, q_{n-1})) \quad \text{[MEDNF-n]}$$

Equations QG-n and MEDNF-n show that the results of macro-executions on the QGSE and the MEDNFSE are identical. In general, the resulting values of q from the QGSE and from the MEDNFSE must be identical since, by definition, the latter is only the disjunctive form of the former. Let us note that this property is obtained by virtue of the definition of the macro execution as a "partially ordered" set of elementary executions "as per the query graph." A set of elementary executions with an arbitrary order would not have this property.

This completes our demonstration of (proof of) equivalence between the query graph and its EDNF representation.

End Example 6 ◊

4.0 EDNF as a Building Block for More Complex Queries

4.1 Incorporating Negation in Horn-Clause Logic

The negation-as-failure evaluation technique [Cla78] extends the linear resolution proof technique by treating a negated literal as a lemma. This lemma is evaluated by a "failure proof." That is, ¬p is inferred if any possible proof of p fails. It has been shown in [Cla78] that the negation-as-failure evaluation technique is a valid inference rule in a *completed database*. Intuitively, a completed database is constructed by adding the "only if" counterpart of the rules in the general form. In the *general form*, rules have no binding in the head (i.e., the rule head has only independent free variables); the bindings are all moved to the rule body (i.e., the right hand side). Also, all rules having unifiable rule heads are combined into one rule by treating the bodies of the individual rules as disjuncts in the body of the new rule.

Clark concludes that negation as failure is valid in the completed database if the rules are limited to those having no variables that appear only in negated literals. In other words, negated literals must be ground before being evaluated. For example, negation as failure is invalid in the following rule:

(1) non-math-major(Student) :- student(Student),
core-math-course(Course), ¬takes(Student, Course, Semester)

Note that the variable Semester does not appear in any positive literal in the rule body.

In practice, however, we have an intuitive interpretation of the rules that do contain free variables in negated literals. We use such rules mainly when we want to express "relational projection" for a negated literal. In this section, we reconcile the use of these rules with a valid construct in Clark's negation as failure. Also, we discuss below safety issues and bottom-up evaluability when negation is involved.

Consider rule (1). Intuitively, we interpret it as saying that a student is not a math major if s/he never took some core math course. Equivalently, we compose a projection of relation "takes" on the first and second attributes and use it in negation. In other words Semester is a don't-care variable. In pure logic, however, the rule does not mean what we want intuitively because the variable Semester can take any value in the domain to make ¬takes(Student, Course, Semester) true. For example, suppose John took all the core math courses: including Calculus in Spring 86. Thus, John is a math major. However, since takes(John, Calculus, Fall 86) is false, rule (1) deduces that John is not a math major.

For comparison, let us consider another set of rules:

(2) non-math-major(Student) :- student(Student),
core-math-course(Course), ¬takes'(Student,Course)

(3) takes'(Stud, Cour) :- takes(Stud, Cour, Semester)

Clearly, these rules provide the same intuitive meaning as that of rule (1). Nevertheless, in this case, Clark's negation as failure, as a

valid inference rule, reflects our intuition correctly. In the completed database, rule (3) is transformed to the equivalent general form as in rule (4):

(4) takes'(S,C) :- ∃*Semester, Stud, Cour* ((S=Stud), (C=Cour), takes(Stud, Cour, Semester))

Subsequently, the *completion law* adds the "only if" part, yielding rule (5) :

(5) takes'(S,C) -: ∃*Semester, Stud, Cour* ((S=Stud), (C=Cour), takes(Stud, Cour, Semester))

Rules (2) and (5) together yield rule (6), which correctly represents our intuition.

(6) non-math-major(Student) :- student(Student), core-math-course(Course), ¬ ∃*Semester, Stud, Cour*((Student =Stud), (Course=Cour), takes (Stud, Cour, Semester))

In our method, we modify any rule in the form of rule (1) to a set of rules in the form of rules (2) and(3), so that the variables in a negated literal not appearing in any positive literal correspond to the attributes that are not projected; thus, conforming to our intuition. This modification is essentially the same as throwing in an existential quantifier for each variable in the negated literal not appearing in any positive literal—leading to a form of rule (6). The form in rule (6) can now be processed using the EDNF approach. Let us note that the modification is exactly the same as what PROLOG implicitly does in the presence of a negated literal containing free variables. In this section, we formalized this implicit modification by reconciling with a valid construct in Clark's negation as failure. A similar technique is used in [Ull87]. Here, the rules in the form of rule (1) are disallowed; instead, the users are required to write rules in the form of rules (2) and (3).

Introducing negation causes many problems regarding safety as well. Consider rule (7):

(7) a(X,Y,Z) :- b(X,Y), ¬c(Y,Z)

Notice that the query ?a(X,Y,Z) will produce a potentially infinite relation even though the relations for b and c are finite. In principle, Z can assume almost any value in the (potentially infinite) domain to make ¬c(Y,Z) true. Furthermore, it is not common to have practically meaningful rules in this form. Thus, we rule out this case by defining the safety criteria as follows:

Definition 1: A rule is *range restricted* [Ban86] if every variable in the head appears somewhere in the body.

For a rule to be bottom-up evaluable, every variable in the body must be "secure" [Ban86], i.e., it cannot assume infinite number of values. We modify the conditions for security of a variable as follows :

Definition 2: A variable in the body of a rule is *secure* if

1. it appears in a positive literal that is not an evaluable predicate, or
2. it is safety dependent [Zan86] (see Section 3.2) on a set of secure variables.

For example in the rule,

a(X,Y,Z) :- b(X,Y), Y=Z,

variables X and Y are secure because they appear in b(X,Y) and Z is secure because it is safety dependent (as defined in Section 3.2) on a secure variable Y. However, in the rule

d(X) :- c(X,L), ¬ e(L,Y), Y=Z,

Z is not secure because neither Z nor Y appears in any positive literal that is not an evaluable predicate.

Finally, the bottom up evaluablility is defined as in [Ban86], but using the new definitions of the term "secure".

Definition 3: A rule is *bottom-up evaluable* if

1. it is range restricted, and
2. every variable in the body is secure.

Definition 4: A query is *bottom-up evaluable* if

1. rules used to construct the query graph are bottom-up evaluable, and
2. negation is not part of a cycle in the query graph.

To process a query, we assume that the query is bottom-up evaluable according to this new definition. In Definition 4, we require negation is not included in a cycle. If negation is involved in a cycle, fixed points cannot be evaluated by an iterative procedure because the relational algebra expressions (functions for which the fixed point is defined) are not guaranteed to be monotone increasing.

4.2 Further Enhancement of the EDNF Approach

EDNF is a building block for more complex queries. In this section, we discuss the processing of more general queries that cannot be represented as a simple EDNF forest. There are two reasons why a simple EDNF is not sufficient.

Case 1: There are cycles that do not pass through the query goal. We call this case *nested recursion*.

Case 2: Negation of a nonleaf node (i.e., the node is not associated with a base relation).

Cycles not passing through the query goal

To explain Case 1, consider the query graph of Figure 14.

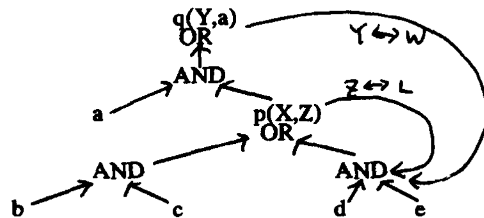

Figure 14. A query graph involving nested recursion.

Due to nested recursion, we cannot represent the query graph in Figure 14 as one EDNF forest. In this case, we construct the query graph treating p as if it were the query goal. Then, we have the EDNF as in Figure 15.

Figure 15. EDNF of Figure 14 using p as the root.

We call p_F the *root of the EDNF forest*. The subscript F identifies such a root. Then, treating p_F as a base relation, the EDNF for q is constructed as Figure 16.

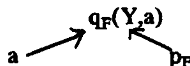

Figure 16. The EDNF for q containing p_F as a leaf.

Then, the evaluation of the query ?q(Y,a) proceeds as follows :

1. Fully evaluate p_F using the EDNF in Figure 15.
2. Evaluate q_F using the EDNF in Figure 16 treating p_F as a base relation.

In general, for a query involving an arbitrary number of recursions placed on different nodes of the query graph, we construct the EDNF graphs as follows :

1. Identify cycles that are interconnected (*strongly connected component* [Mor86]).
2. Identify a set of (goal) nodes such that each cycle contains at least one node in the set. We define such a set as a *cover* (or *feedback set* [Gar79]) of the strongly connected component.
3. Construct an EDNF forest for each node in the cover set with that node as the root of the EDNF. For the EDNF of a particular node, the other nodes in the cover set are treated as base relations.
4. Construct the EDNF of the query goal (if not already in the cover set) treating all the nodes in the cover set as base relations.

Note that, for complicated query graphs with interconnected cycles, the choice of the cover set itself is complex. The cover set with minimum cardinality is called a *minimal cover set*. Our strategy for general recursive query processing is then to choose a minimal cover set for each strongly connected component. Each node in the cover set is assigned as the root of a separate EDNF forest. Then, the resulting forest of trees, where leaf nodes themselves may be EDNF forests, is processed by obeying an *EDNF forest dependency graph*, which establishes the partial order of execution of the EDNF forests. For example, the forest dependency graph for EDNF forests in Figure 14 is shown in Figure 17 .

Figure 17. Forest dependency graph for the query in Figure 14.

Figure 17 says that EDNF forest p_F should be evaluated before q_F . We have developed heuristics for identifying cover sets and are currently investigating alternative solutions to the minimization of cover sets.

Note that an important characteristic of our approach is that we evaluate only the minimal number of nodes (i.e, the cover set) that are absolutely necessary for processing the query. In contrast, in other methods [Mor86], *all* the nodes in the original query graph must be evaluated.

If the cover set contains more than one node, execution of the strongly connected component involves the execution of each EDNF forest in that component in turn until a fixed point is reached. Detailed algorithms for the construction of the dependency graph of EDNF forests and their processing in the general case exists, but they are beyond the scope of this paper.

Negation of nonleaf nodes

If the query has negation on a leaf node in the query graph, an EDNF structure can be composed for the query as in Section 3. If it has negation on a nonleaf node, then we construct a separate EDNF with the negated node as the root. The EDNF of the original query is then constructed treating the negated EDNF as if it were a leaf node. Then, the technique of decomposing a query into multiple EDNF forests would apply. Figure 18 shows an example of the query graph involving negation and its EDNFs. This strategy shares the same concept as stratification [Apt85] or layering [Naq86a]. Stratification relaxes the hierarchical condition originally proposed by Clark [Cla78] by allowing recursion not involving negation.

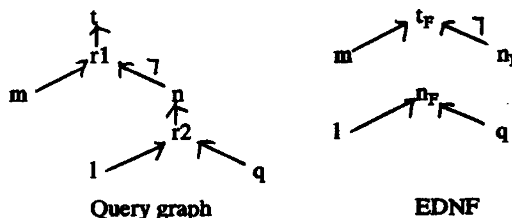

Figure 18. Query graph involving negation and its EDNFs.

Controlling the number of disjuncts

One possible drawback of the EDNF transformation is the potential existence of too many disjuncts (trees in the EDNF) with overlapping information among one another. We solve this problem by not expanding important branches of the query graph and designating them as separate EDNF forests. Thus, the EDNF of the original query is composed treating these branches (separate EDNF forests) as relations. Important branches are those that may be duplicated in many trees when expanded and that would be expensive to process. By using this technique we can prevent proliferation of trees in the EDNFs.

5.0 Advantages of the EDNF Approach

The EDNF approach to processing logic queries has the following advantages:

1. Use of existing DBMSs:

   Our primary thrust in the approach is to exploit the facilities within existing DBMSs as much as possible. This means: 1) No modification to the DBMS is necessary; 2) Facilities provided by the DBMS, such as authorization, cataloging, recovery, etc., may be utilized; 3) Entire query optimization techniques for relational databases with their advantages can be harnessed.

2. Performance:

The EDNF approach enhances performance in various ways:

- First, when compared with naive loose coupling, it obviously reduces the number of calls to the DBMS: it avoids issuing a large number of requests for a small amount of data by essentially batching them in large database queries.
- Second, the EDNF eliminates the arbitrary processing structure imposed by the user-written rules through normalization and keeps only the information essential to process the query. This normalization provides the DBMS optimizer with more flexibility in finding the best access plan [Kim82].
- Third, the EDNF minimizes the need to create temporary relations by concentrating only on the desired results and avoiding to create unnecessary intermediate results. In contrast, many conventional methods try to evaluate all the intermediate results and store them in temporary relations. Evaluating all these intermediate results could be expensive as explained in Section 2.

3. Better understanding of recursive logic queries:

The EDNF serves as a clear characterization of a large class of logic queries. We have shown that the queries can be represented in a small number of primitive constructs (i.e., the nonlooped tree, single-looped tree, and multilooped tree). Further, the EDNF provides a graphical classification of queries and makes it easy to *visualize* the complexity of recursion. For example, any query composed from a linear system of rules produces an EDNF forest with only nonlooped and single-looped trees. On the other hand, any query from a nonlinear system of rules produces at least one multilooped tree. Other types of recursions can be captured easily. For example, a conventional simple transitive closure appears as one nonlooped tree and one single-looped tree each having one leaf. We also have shown in Section 3.2 that any query constructed from a linear set of rules can be viewed as a transitive closure of a complex function.

4. Availability of alternative processing algorithms:

Due to the characterization the EDNF provides, we can construct a variety of algorithms for different compositions of EDNF trees. For example, in addition to the basic algorithm for processing any EDNF, we have illustrated an efficient algorithm for processing an EDNF forest with single-looped and nonlooped trees (derived from linear rules). This algorithm avoids duplicate processing by implicitly calculating the differentials. Thus, it satisfies the FRD-B property. Unlike the ones previously reported, this algorithm does not require explicit relational algebra expressions for the differentials. We are also contructing algorithms for other specialized situations. The availability of these algorithms allows an high-level optimizer to choose the best one for a specific situation.

6.0 Summary

We have proposed an approach to processing logic queries based on the Extended Disjunctive Normal Form (EDNF). The main purpose of this approach is to support deduction with existing DBMSs in a loosely coupled manner. The class of queries considered encompasses those in function-free Horn-clause logic extended for negation. For safe evaluation, however, we limit the scope to bottom-up evaluable rules per our new definition.

We have presented the EDNF and its processing algorithm. We have formally proved the correctness by showing that the evaluation of the query based on the EDNF is identical to the results obtained by a conventional method using the rule/goal graph.

In our opinion, virtually none of the current work on logic query optimization available in the literature deals with negation in a practical manner. We have proposed a technique of extending Clark's *negation as failure* to include cases that are practically important and incorporated it in our general query processing algorithm. We have also defined new criteria for safety and termination in the presence of negation. In particular, the definitions of security of the variables and bottom-evaluability of the rules have been revised for negation.

As a future research, we left the superimposition of the FRD-A property on our approach as an open issue. We are evaluating existing techniques, particularly magic sets and counting, for a possible incorporation into our framework.

Although many techniques have been proposed for logic query processing, not much has been reported for application to loosely coupled environments to exploit already existing DBMSs. We believe that our technique provides significant progress in this direction.

Acknowledgement

We would like to thank Stephen Brady for clarifying concepts in theory and predicate calculus. Brady deserves the credit for initiating the SIENA project, of which the SQL Inference Engine is a part. Andy Kaplan and Tetsuya Furukawa contributed by implementing part of SQL Inference Engine. We acknowledge Bob Paige for giving us formal statements and proofs of the theorems and lemmas regarding fixed points in Section 3. Jeff Jaffe contributed some crucial comments. Discussions with Michael Kifer, Laurent Vieille and Jean-Marie Nicolas were useful. Finally, we are very grateful to Aruna Navathe for the assistance in manuscript preparation.

REFERENCES

[Agr87] Agrawal R.K., "ALPHA : An Extension of Relational Algebra to Express a Class of Recursive Queries," *Third Intl. Conf. on Data Engineering*, Los Angeles, CA, Feb. 1987.

[Aho79] Aho, A. and Ullman, J., "Universality of Data Retrieval Languages," in *Proc. 6th Symposium on Principles of Programming Languages*, pp. 110-117, Jan. 1979.

[Apt85] Apt, K.R., Blair, H. and Walker A., "Towards a Theory of Declarative Knowledge," unpublished manuscript, 1985.

[Ban85] Bancilhon, F., "Naive Evaluation of Recursively Defined Relations," in *On Knowledge Base Management*

Systems–Integrating Database and AI Systems, (eds. M. Brodie and J. Mylopoulos), Springer Verlag, 1985.

[Ban86] Bancilhon, F. and Ramakrishnan, R., "An Amateur's Introduction to Recursive Query Processing Strategies," in *Proc. Intl. Conf. on Management of Data*, pp. 16-52, 1986.

[Ban86a] Bancilhon, F. et al., "Magic Sets and Other Strange Ways to Implement Logic Programs," *Proc. ACM Symp. Principles of Database Systems*, 1986.

[Boc86] Bocca, J., "EDUCE-A Marriage of Convenience : Prolog and a Relational DBMS," in *Proc. Third Symposium on Logic Programming*, Salt Lake City, Utah, 1986.

[Boc86a] Bocca, J. et al., "Some Steps Towards a DBMS Based KBMS," *Information Processing*, 1986.

[Cai87] Cai, J. and Paige, R., "Binding Performance at Language Design Time," in *Proc. ACM Symp. on Principles of Programming Languages*, pp. 85-97, 1987.

[Cer86] Ceri, S., Gottlob, G., and Wiederhold, G., "Interfacing Relational Databases and Prolog Efficiently," *Proc. First Expert Database Systems Conference*, Charleston, SC, pp. 141-153, April 1986.

[Cha85] Chang, C.L. and Walker, A., "PROSQL : A PROLOG Programming Interface with SQL/DS," in *Proc. Expert Database Systems Workshop*, Charleston, SC, 1985.

[Cla78] Clark, K., "Negation as Failure" in *Logic and Databases*, (eds. H. Gallaire, J. Minker, and J.Nicolas), Plenum Press, 1978.

[Dev86] Devanbu, P. and Agrawal, R., "Moving Selections into Fixpoint Queries," Working Paper, A.T. and T. Bell Laboratories, 1986.

[Gar79] Garey, M. and Johnson, D., *Computer and Intractability*, Freeman, 1979.

[Hen84] Henchen, L.J. and Naqvi, S.A., "On Compiling Queries in Recursive First-Order Databases," *Journal of the ACM*, Vol. 31, No. 1, pp.47-85, Jan. 1984.

[Ioa86] Ioannidis, Y.E. and Wong, E., "On the Computation of the Transitive Closure of Relational Operators," *Twelfth Intl. Conf. Very Large Data Bases*, Kyoto, Japan, pp.403-411, Aug. 1986.

[Jag87] Jagadish, H.V. and Agrawal, R., "Computing Linear Recursion as Transitive Closure," in *Proc. Intl. Conf. on Management of Data*, San Francisco, (to appear), May 1987.

[Jar84] Jarke, M., Clifford, J., and Vassiliou, Y., "An Optimizing Prolog Front-End to a Relational Query System," in *Proc. Intl. Conf. on Management of Data*, pp.296-306, 1984.

[Kif86] Kifer, M. and Lozinskii, E.L., "A Framework for an Efficient Implementation of Deductive Databases," in *Proc. 6th Adv. DB Symposium*, Tokyo, Japan, 1986.

[Kif86a] Kifer, M. and Lozinskii, E.L., "Filtering Data Flow in Deductive Databases," in *Proc. Int. Conf. on Database Theory*, Rome, Italy, Sept. 1986.

[Kil73] Kildall, G., "A Unified Approach to Global Program Optimization," in *Proc. ACM Symp. on Principles of Programming Languages*, Oct. 1973.

[Kim82] Kim, W., "On Optimizing a SQL-like Nested Query," *ACM Trans. Database Syst.* Vol. 7, No. 3, pp. 443-469, Sept. 1982.

[Kri86] Krishnamurthy, R. and Zaniolo, C., "Safety and Optimization of Horn Clause Queries," in *Proc. Foundation of Deductive Databases and Logic Programming* (ed. J. Minker), 1986.

[Loz85] Lozinskii, E. L., "Evaluating Queries in Deductive Databases by Generating," in *Proc. 9th Intl. Joint Conf. on Artificial Intelligence*, Los Angeles, 1985.

[Mck81] McKay, D. and Shapiro, S., "Using Active Connection Graphs for Reasoning with Recursive Rules," in *Proc. Seventh Intl. Joint Conf. on Artificial Intell.*, Vancouver, B.C., Aug. 1981.

[Mor86] Morris, K., Ullman, J., and Van Gelder, A., "Design Overview of the NAIL System," *Proc. 3rd Intl. Conf. on Logic Prog.*, London, July, 1986.

[Naq86] Naqvi, S., "Negation as Failure for First-Order Queries," in *Proc. ACM Symp. on PODS*, Cambridge, Mass., pp. 114-122, Mar. 1986.

[Naq86a] Naqvi, S., "A Logic for Negation in Database Systems," in *Proc. Foundation of Deductive Databases and Logic Programming*, (ed. J. Minker), Washington, D.C., pp.378-387, Aug. 1986.

[Nic86] Nicolas, J.M., *Personal Communication*, November 1986.

[Rei78] Reiter, R., "On Closed World Databases," in *Logic and Databases* (eds. H. Gallaire and J. Minker), Plenum Press, 1978.

[Sac86] Sacca, D. and Zaniolo, C., "The Generalized Counting Method for Recursive Logic Queries," in *Proc. Int. Conf. on Database Theory*, Rome, Italy, Sept. 1986.

[Sci85] Sciore, E. and Warren D.S., "Towards an Integrated Database-Prolog System," in *Proc. Expert Database Systems Workshop* Charleston, SC, pp.293-305, 1985.

[Tar85] Tarski, A., "A Lattice Theoretical Fixpoint Theorem and Its Applications," *Pacific Journal of Mathematics*, Vol. 5, pp.285-309, 1985.

[Ull85] Ullman, J., "Implementation of Logical Query Languages for Databases," *ACM Trans. Database Syst.*, Vol. 10, No. 3, pp. 289-321, Sept. 1985.

[Ull87] Ullman, J., *Unpublished manuscript*, 1987.

[Van86] Van Gelder, A., "A Message Passing Framework for Logical Query Evaluation," in *Proc. Intl. Conf. on Management of Data*, Washington, D.C., pp. 155-165, May 1986.

[Vie86] Vieille, L., "Recursive Axioms in Deductive Databases : The Query/Subquery Approach," in *Proc. First Expert Database Systems Conference*, Charleston, SC, pp. 179-193, Apr. 1986.

[Wal85] Walker, A., "SYLLOG : An Approach to PROLOG for Non Programmers," *Logic Programming and its Applications*, (eds. M. Van Caneghem and D.H.D. Warren), Ablex, Norwood, NJ, 1985.

[Wha85] Whang, K.-Y., "Query Optimization in Office-by-Example," IBM Research Report RC 11571, Dec. 1985.

[Wha87] Whang, K.-Y. and Navathe, S., *An Extended Disjunctive Normal Form Approach for Processing Recursive Logic Queries in Loosely Coupled Environments*, IBM Res. Rep. RC12567, Mar. 1987.

[Yu87] Yu, C.T. and Zhang, W., "Efficient Recursive Query Processing Using Wavefront Methods," *Third Intl. Conf. on Data Engineering*, Los Angeles, CA, Feb. 1987.

[Zan86] Zaniolo, C., "Safety and Compilation of Non-Recursive Horn Clauses," in *Proc. First Expert Database Systems Conference*, Charleston, SC, pp. 167-178, Apr. 1986.

[Zlo77] Zloof, M.M., "Query-by-Example: A Database Language," *IBM Systems J.*, Vol. 16, No. 4, pp. 324-343, 1977.