The Representation and Deductive Retrieval of Complex Objects

Carlo Zaniolo
MCC
Austin, Texas, U.S.A.

ABSTRACT

The Relational Data Model and Relational Calculus are extended with Unification and non-recursive Horn Clauses from Logic. The benefits gained include better versatility and a richer functionality for expressing complex facts, deductive queries and rule-based inferences. Applications include semantic data models for Databases, frames for Knowledge-based systems, and Complex Objects for CAD. An Extended Relational Algebra (ERA) is introduced that has the same expressive power as the new Calculus. The algorithm given for translating from Calculus to ERA supplies a sound basis for the compilation of these Horn clauses, and their implementation using query optimization and other techniques currently used in database systems.

1. Introduction

A strong interest is emerging in combining Databases and Logic Programming [Gallaire 84, Jarke 84, Li 84, Parker 85, Ullman 85], for reasons of theoretical and practical import. From the theoretical viewpoint, there is a deep affinity between relational databases and logic programming, resulting from their common ancestry of mathematical logic. From the practical viewpoint, there is the realization that the two technologies provide complementary benefits. Logic Programming (LP) entails an expressive power which greatly surpasses that of current database query languages, as demonstrated by its use in applications as varied as deductive retrieval, translator systems, CAD and expert systems. Databases, on the other hand, provide the technology for storing, managing and processing efficiently very large data banks, in addition with amenities such as sharing, recovery and security. Therefore, it appears that a combination of the two technologies holds the promise for new data and knowledge base systems of great power and versatility. This paper contributes towards such an objective by extending relational database techniques to support unification on databases of complex facts.

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.

2. Databases and Logic

In a Logic Language, the following facts can be used to represent information about the first names, the last names, the ages and the fathers of individuals.

person(david, smith, 55, john)
person(jane, smith, 22, david)
person(frank, green, 23, travis)

Figure 1: A person fact base.

These simple facts can be represented in a relational database using a four-column relation **person** containing three tuples.

Queries against a fact database can be expressed in a style similar to a in-line version of QBE [Zloof 75]. For instance, let us use initial lower case to denote constants, initial upper case for variables, and the symbol "_" for *anonymous variables* (i.e., don't care). Then, we can express the query to find all persons with last name "Smith" as follows:

{<Nm, Ag> | person(Nm, smith, Ag, _)} ?

Figure 2. Find names and ages of persons.

Such a query, will cause our LP system to return *all the pairs* satisfying it:

<david, 55>
<jane, 22>

Applications, in a LP language are developed via extensive use of *rules*. For instance, a rule to define the paternal grandfather of a person can be written as

follows:

grandpa(Young, LN, Old) ←
person(Young, LN, _, Middle) &
person(Middle, LN, _, Old)

Figure 3. The paternal grandfather is the father of the father.

Rules can be thought of as a generalization of the derived view mechanism found in relational systems, which provide for their efficient implementation via the usual mechanisms for query support. These mechanisms are often based upon the mapping of queries into equivalent Relational Algebra expressions [Ullman 80]. For instance, our grandpa view can be translated into a join of the second and fourth column of **person** with the second and first column of **person** again, followed by a projection. Indeed, Codd's fundamental result [Codd 71] states that any query of relational calculus can be translated into an equivalent expression of relational algebra. The importance of this result cannot be overemphasized since, (i) it characterizes the expressive power of relational systems, and (ii) it supplies the foundation for efficient implementations of relational queries by compilation and optimization techniques or special hardware (database machines).

No relational algebra equivalent is currently known for Horn Clause Logic, since this surpasses the expressive power of relational calculus in two main respects:

(1) the availability of recursive rules, and

(2) the availability of general unification.

The subject of databases with recursive rules has received wide attention, particularly in relation to rule compilation, and replacement of recursion with iteration [Aho 79, Chandra 82, Henschen 84, McKay 81, Naqvi 83, Ullman 85]. Comparatively less attention has been paid, until now, to the unification problem addressed in this paper. In the next section we illustrate, by means of examples, the many useful applications of unification in databases and knowledge-based systems.

3. Complex Facts and their Applications

The arguments of a logic predicate need not be simple, but they may be complex— i.e., consist of a functor with possibly complex arguments. A great deal of power emanates from this flexibility. Say, for instance, that we want to store information about employees, including their last and first names, their job classification and their education. The information requirements for education depend on the education level. For instance, only the graduation year is stored for high school graduates, while a degree description, with major and graduation year, is kept for college graduates. For bachelor and higher degrees the school name is also required. A sample from our database is shown in Figure 4.

```
%       a relation of facts with complex fields:
%       emp(FirstName, LastName, JobClass, Degree)
emp(joe, cool, porter,    none)
emp(max, fax, guard,      degree(hs, 1976))
emp(joe, doe, vp,         degree(ms, engl, school(harvard, ma), 1981))
emp(fred, red, staff,     degree(ms, ba, school(usc, ca), 1983))
```

Figure 4. A database of complex facts.

Thus, while Joe Cool has no degree, as indicated by "none", Max Fax got his high school degree in 1976. Moreover, Joe Doe got a Master in English from Harvard, MA, in 1981, and Fred Red got his Master in Business Administration from USC, CA, in 1983. (Note the use of a function symbol with different numbers of arguments. Although this is a departure from the textbooks on Logic, it is commonly done in Logic Programming.)

In Logic, queries and rule-based deduction against a database of complex facts, such as that of Figure 4, can be expressed in a very simple fashion. For instance, to find all the MBA's, with their schools, who graduated after 1981, one only needs to state the rule:

new-mbas(LN, FN, Sch, Year) ←
emp(FN, LN, _, degree(ms, ba, Sch, Year)) &
Year >1981

Figure 5. Name, School and Year for MBAs who graduated after 1981.

Then the query:

{<L,F,S,Y> | new-mbas(L, F, S, Y)} ?

Figure 6. Retrieving the MBAs of Figure 5.

will return:

<red, fred, 1983, school(usc, ca)>.

These examples illustrate the flexibility and power of Logic Programming in dealing with complex facts, inasmuch as the subcomponents of a term can be qualified separately or treated as a unit (e.g., the argument **Sch** in **degree**). To come close to this kind of flexibility in a more conventional DBMS, the concept of generalization [Smith 77] will have to be supported; then two subtypes must be declared, one for employees with high school degree only, the other for employees with a college degree. As discussed in [Zaniolo 83] queries involving generalization can be handled by extensions of relational languages. These in turn can be implemented on top of relational systems [Tsur 84], but not without additional complications such as extensive use of null values. However, complex objects in a LP system provide the power of semantic data models in a simpler and more versatile framework — complex objects can either be made available to users directly, or can be used in the implementation of a semantic data model and language, such as that described in [Zaniolo 83].

For a second example, let us assume that we have a list of names and states for Ivy League universities:

```
%      name and state of ivy league schools
ivy  (harvard,      ma)
ivy  (princeton,    nj)
ivy  (brown,        ri)
ivy  (yale,         ct)
ivy  (cornell,      ny)
ivy  (pennsylvania, pa)
ivy  (columbia,     ny)
ivy  (dartmouth,    nh)
```

Figure 7. The old Ivy League.

A list of all Ivy League graduates, for the 80's can be obtained as follows:

```
% ivyup stands for Ivy_Yuppies
ivyup(Ln, Fn, Yr) ←
emp(Fn, Ln, _, degree(_, _, school(ScN, StN), Yr))
& ivy(ScN, StN) & Yr > 1979 & Yr < 1990
```

Figure 8. Defining those Ivy League Yuppies.

Another very useful application of complex objects is for knowledge primitives such as frames, whose representational power is based upon flexible mechanisms for deriving values associated with arguments (slots) [Winston 79]. For instance, values could be explicitly stored or derived via calculations or default assignments. The functionality of frames can be supported using facts and rules with complex arguments [Zaniolo 84][Kowalski 84] as illustrated by the following example.

We have flat CAD parts identified by a unique part number and described by their geometric shape. Thus circles are characterized by their diameter (one parameter), while rectangles are described by their base and height (two parameters) and triangles by their three sides (three parameters), and so on. There is also a third argument that defines the weight of the part. A sample database could for instance be the following.

```
% part(Pno,     Shape,                 Weight_Info)
part(0011, rectangle(11., 7.),   value(140.))
part(1002, triangle(4., 3., 5.), negligible)
part(1033, square(23.5),         default)
part(2000, circle(30.),          table)
part(2222, circle(30.),          table(2000))
```

Figure 9. These are flat parts.

The third arguments of these five facts illustrate five different schemes for deriving the weight of the part.

value(140.): The part's measured weight is 140 (onces); this value can thus be stored explicitly.

negligible: The weight of the part is minimal (and can be regarded as zero in most situations);

default: The weight of the part has not been measured but it can be estimated by multiplying its surface area (known from its shape) by its specific weight.

table: This is a prototypical part whose weight is kept in a table called w-table.

table(2000): The weight for this part is to be found by looking up the weight of the prototypical part 2000.

These five different schemes for computing the weight of a part can be implemented by the following five rules:

```
weight(Pno, W) ←  part(Pno, _, value(W))
weight(Pno,  0) ← part(Pno, _, negligible)
weight(Pno, W) ←  part(Pno, Shape, default)
                  & area(Shape, Area)&
                  W = Area* 1.2
weight(Pno, W) ←  part(Pno, _, table)&
                  w-table(Pno, W)
weight(Pno, W) ←  part(Pno, _, table(P2))&
                  w-table(P2, W)
```

Figure 10. Five ways to get the weight of a flat part.

These rules in turn, refer to other rules for computing areas,

area(rectangle(Base, Height), Ar) ←
Ar = Base * Height
area(square(Base), Ar) ← Ar = Base * Base

Figure 11. Area rules for geometric shapes.

and to a list of prototypical weights, such as:

w-table(1221, 12.5)
w-table(1136, 131.6)
w-table(2000, 25.6)

Figure 12. The weight table.

Thus complex arguments and associated rules deliver the functionality of frames (not shown here, is the support for the *ISA* inheritance which can be obtained with the techniques described in [Zaniolo 84]). Applications may either use rules and complex facts directly, or through an additional layer that supports frames as a high level construct — internally implemented by suitable rules on complex objects.

Another very important use of unification and complex arguments is in the representation and handling of recursive data structures, such as trees and lists. It is also well-known that functions symbols cannot be removed from recursive predicates by the introduction of additional predicates. Now, while the operators described in the next sections are extremely useful in processing efficiently these recursive data structures as well, a discussion of this topic is not to be found in this paper, since treating it would force us into the difficult problem of recursive rule compilation, which is orthogonal to the scope of this manuscript that is primarily concerned with unification.

4. An Extended Relational Algebra

The execution strategies for queries in Logic Programming system are very different from those of a database system. In the current implementations of a logic based language, e.g., Prolog's interpreters and compilers [Campbell 84], the processing of rules is intertwined with the accessing of facts. Moreover, the results of successful unifications are kept as variable bindings in the environment. Alternative solutions are produced through backtracking. In relational database systems however, a query on either base relations or on derived views, would be compiled into sequences of relational algebra-like operators, that are later executed against the fact database. For instance the query,

{<Gc, Lst, Gp> | grandpa(Gc, Lst, Gp)}?

can be translated into an equivalent relational algebra tree:

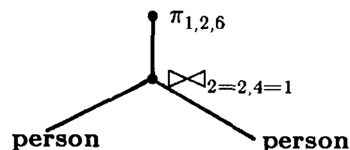

Figure 13. Relational Algebra tree for of Figure 3.

Such a tree is commonly used by a query optimizer to generate the actual program that implements the query. In addition, it outlines a conceptual execution model for compiled database queries. In this model, data is viewed as flowing upwards from the leaf nodes toward the top, and being transformed in the process by the operators associated with the nodes. In contradistinction with Prolog's execution model, this execution paradigm is denoted by

i) the absence of explicit variable bindings since all inter-rule unification work is done at compile time (intra-rule unifications, i.e., joins, are still performed at execution time).

ii) the absence of backtracking, since an explicit set of tuples (partial results) is now associated with each node. These partial results are then passed to the ancestor nodes, possibly as they are being generated — on the fly — for select and project nodes, or after some off-line storing and processing in the case of joins.

As realized by various researchers [Reiter 78, Ullman 85], the database execution strategy offers many advantages in terms of efficiency, particularly when large fact bases are involved, since much of the unification work is done at compile time, and a set-oriented processing is used rather than a tuple-at-a time processing with backtracking. Thus, a number of *optimization and special-purpose techniques can* be applied to speed up execution. For instance, sort-joins can replace the looping-join strategy to which Prolog's compilers are restricted. (Our arguments are couched here in terms of software implementations but similar considerations can be made for hardware.)

Therefore, we want to provide database-like techniques for compiling logic queries involving unification, such as that of Figure 4, where we have conditions involving subcomponents of complex arguments. Since the standard relational algebra is inadequate to the task, we introduce an *Extended Relational Algebra (ERA)* that will do the job. This algebra contains the usual set union, and Cartesian product, plus *extended select* and *project* operators and a new operator called the *combine*.

Extended Select: When operating on traditional simple arguments, the new select operator reduces to the old relational one. For complex arguments, however the new operator can select on the basis of the value of both functors and subarguments. We use

Dewey's decimal notation, to denote the subarguments in a complex object.[1] Thus the operator:

$$\sigma[4=none]$$

and

$$\sigma[4.1=ms, 4.4>1981]$$

will respectively select from **emp** those with no degree and with an MS degree after 1981. Note that complex objects can *either be used as a unit or addressed by individual subcomponents.* Thus,

$$\sigma[4.3=school(harvard, ma)]$$

and

$$\sigma[4.3.0=school, 4.3.1=harvard, 4.3.2=ma]$$

represent two legal ways to retrieve Harvard's alumni from the **emp** relation. We can also select on the value of the functor of a complex argument, which will be denoted by a zero suffix. Thus,

$$\sigma[4.0=degree]$$

may be used to select from **emp** all those employees who have a degree. The argument count in a complex term T will be denoted by $T.count$. This allow us to express selection on the number of arguments of T (see next section for a formal definition of the number of arguments in a term). For instance, a query to find a tuple in **emp** where the fourth argument contains exactly 2 subarguments, can be as follows:

$$\sigma[4.count=2]$$

Thus, only the second tuple in Figure 4 satisfies this condition. This kind of condition will be called an *arity condition.*

Extended Project: The *extended project* operator also refers to arguments and subarguments using Dewey's notation. For instance,

$$\pi[1,2,4]$$

will retrieve all **emp** tuples, with the third argument omitted, much as in the traditional relational style. However, the operator

$$\pi[1,2,4.1,4.3]$$

applied to emp will return

<joe, doe, ms, school(harvard, ma)>
<fred, red, ms, school(usc, ca)>

thus eliminating the first two tuples for which the component 4.3 is not defined.

For convenience, we also define two additional operators. The first is the traditional (equi)join, that can be defined as a Cartesian product followed by equality restrictions. We follow here the convention of not repeating the join columns; thus, in Figure 3, two instances of **person**, that has arity four, joined by equating two pairs of columns, yield a relation of arity six. We also follow the convention of considering the Cartesian product a degenerate type of join. The second derived operator is the Extended Select/Project (ESP), described next.

Extended Select/Project (ESP): This operator is defined as the composition of an extended select followed by an extended project. For instance, the composition of

$$\sigma[4.3.1=harvard, 4.3.2=ma]$$

with

$$\pi[1,2,4.1,4.3]$$

is denoted by,

$$\rho[4.3.1=harvard, 4.3.2=ma\ /1,2,4.1,4.3]$$

We will refer to the lists to the left and right of the slash as the σ*-list* and the π*-list*, respectively.

Combine Operator: The combine operator combines the various arguments of a predicate into one or more *complex arguments.* It will be denoted by gamma. For instance,

$$\gamma[school(1, state(2))]$$

applied to the ivy relation, produces tuples of the form:

<school(harvard, state(ma))>

Thus, the expression in the brackets of a combine is basically a complex fact, where integers are used to denote arguments by their position number (if the number exceeds the argument count in the relation, the result is undefined). Thus:

$$\gamma[a(1,2), b(3)]\ \rho[1.0=a, 2.0=b\ /1.1,1.2,2.1]$$

reduces to the identity operator for any relation with three or more arguments.

[1] Take, for instance, the last fact in Figure 4. Argument # 3 denotes "staff", while 4 denotes "degree(ms, ba, school(usc, ca), 1983)". Thus 4.1 and 4.2 respectively denote "ms" and "ba", while 4.3 and 4.4 respectively denote "school(usc, ca)" and 1983. Thus, 4.3.1 and 4.3.2 denote "usc" and "ca", respectively.

For convenience, we also allow the combine operator to reorder and repeat its arguments. Thus

$$\gamma[2,1,3,1]$$

also represents a valid combine.

In summary, we have defined an Extended Relational Algebra (ERA) containing the following operators: *ESP, join, combine* and *union*. The importance of these operators follow from the following observations:

A) Non-recursive, safe Horn Clauses can be compiled into ERA expressions, and

B) ERA expressions can be supported efficiently using relational database implementation techniques.

Before we enter a formal discussion, let us illustrate these points with some examples. The query of Figure 6 can be translated into either of the following trees:

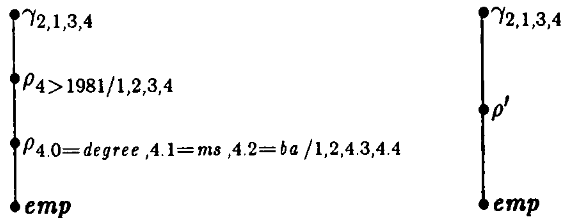

Figure 14. Two equivalent ERA trees for the query of Figure 6, where ρ' denotes:
$\rho_{4.0=degree,4.1=ms,4.2=ba,4.4>1981/1,2,4.3,4.4}$

On the right, the two ESP operators were combined into one — the composition of two ESP operators is always another ESP operator. The translation for the rule of Figure 8 is instead:

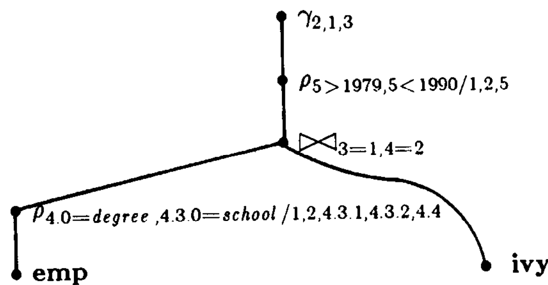

Figure 15. An ERA tree for the rule of Figure 8.

This example uses the combine operator in the simpler form, where only a reordering of arguments takes place.

5. Non Recursive Safe Formulas

Let us formalize the previous examples by defining a Horn Clause Logic containing only non-recursive, safe formulas.

We can start with the definition of our basic tokens (different conventions are possible here).

Tokens: any string without reserved characters: space, comma, parentheses, braces, ampersand, left arrow, question mark.

Numbers: The usual integer, floating point, and scientific notations will be supported.

Variables: Tokens which begin with an upper case letter denote variables.

Terms: A term is either a token or has the form

$$f\ (arg_1,\ldots,arg_n)$$

where:

f, that syntactically must be a token other than a number or a variable, is called a *functor,* and

$arg_1,\ldots,arg_n$ are terms, called the arguments of $f\ (arg_1,\ldots,arg_n)$.

A term will be called *simple* when all its arguments are tokens; it will be called *complex* otherwise. A variable of a term is one that appears as one of its arguments, or as a variable in some argument of the term.

Predicates: Predicates are top-level terms — i.e., terms that are not arguments of any other term.

There is a number of built-in comparison predicates. These have the form: $X@Y$, where @ denotes a comparison operator: $=$, $\neq$, $>$, $<$, $\geq$, $\leq$, and X and Y are tokens. (The $>$ and $<$ test always fails if X or Y is not a number).

Facts: A (simple/complex) predicate without variables is called a (simple/ complex) fact.

Tuples: The vector of the arguments of a (simple/complex) fact will be called a (simple/complex) tuple.

Rules: A rule has the form:

$$Q \leftarrow P_1 \& \cdots \& P_k \tag{5.1}$$

Where Q, $P_1,\ldots, P_k$ denote predicates. Q is called the left side, or the head, of the rule, while the conjunction $P_1 \& \cdots \& P_k$ is called its right side, or tail.

In a database of rules, the notion of predecessors of a predicate can be characterized as follows:

(i) Q is the predecessor of P if Q is the head of the rule while P appears on the right side.

(ii) The predecessor relation is ***reflexive*** and ***transitive***.

Non-Recursive Rules:
A rule where the predicate on the left side is not a predecessor of any predicate on the right is called non-recursive.

Given a database of facts and rules we can define the notion of safety as follows:

Safety: A predicate that only unifies with database facts is safe.
A rule is safe when all its variables appear in some safe predicate (safe variables).
A predicate that only unifies with safe rules is also safe.

Queries: Queries have the form:

$$\{<X_1,...,X_m> \mid Q_1\&...\&Q_k\}? \tag{5.2}$$

The result of such a query is defined as the set of all m-vectors of values that satisfy $Q_1\&...\&Q_k$ in the usual logic sense.

Query (5.2) will be called *safe* when $Q_1, \cdots, Q_k$ only unify with either facts or safe rules.

A set of facts, non-recursive safe rules, and safe queries will be called a *NRS Database*. It will be proven later that the safety conditions above do in fact guarantee that answers to queries in a NRS database are finite. We will use the term *relation* to denote the set of facts having the same functor and the same number of arguments.

The intensional information in a NRS Database can be represented by a predicate connection graph [Kowalski 75, McKay 81]. For notational convenience, queries in this graph, will be assimilated to rules derived as follows. A query, such as (5.2), will be represented by a rule

$$query_j(X_1,...,X_m) \leftarrow Q_1\&...\&Q_k \tag{5.3}$$

Where $query_j$ is the unique name assigned to this particular query.

The *Predicate Connection Graph* [Kowalski 75, McKay 81] is the (directed) and/or graph constructed as follows:

Rule nodes: The head of each rule is denoted by an and-node.

Goal nodes: Each predicate in the tail of each rule is denoted by an or-node.

Relation nodes: Each relation is denoted by an and-node.

Edges in our directed graph are constructed as follows. From each rule node, there are edges to all predicates (goal nodes) in the tail of the rule. Moreover, for each goal node G, either there are edges from G to every rule node that *unifies* with it, or there is an edge from G to a relation node R, where R has the same functor and number of arguments as G.

Example 5.1: This consists of (i) the relations **emp** and **ivy** of Figures 4 and 7, and of the rules and queries given in Figure 16. Observe, that in addition to the rules and queries of Figures 5,6 and 8, we now have two new rules defining the procedure **wsj**, and a query on this.

Example 5.1 defines an NRS Database with a predicate connection graph given in Figure 17. In our predicate connection graph, edges are implicitly assumed to point downwards. Note that each rule node is labeled by a left arrow. Goal nodes are their parents or children. Leaf nodes that are also goal nodes denote comparison predicates, and leaf nodes that are not goal nodes denote relations. Each edge from a goal node to a rule node defines a unification for the corresponding arguments of the two nodes. Unlike in the Rule/Goal Trees described in [Ullman 85] however, the unification results are not propagated throughout the tree. The following property follows directly from the definitions:

Lemma 5.1. The predicate connection graph for a NRS database is acyclic.

```
% QUERIES
{<L,F,S,Y> | new-mbas(L, F, S, Y)} ?
{<L, F, Grad> | wsj(L, F, Grad)} ?
% RULES
% wsj stands for Wall-Street Journal subscribes
wsj(Last, First, mba(Yr)) ←
        new-mbas(Last, First, _,Yr)
wsj(Last, First, ivylg(Yr)) ← ivy(Last, First, Yr)
new-mbas(LN, FN, Sch, Year) ←
emp(FN, LN, _, degree(ms, ba, Sch, Year)) &
        Year >1981
ivyup(Ln, Fn, Yr) ←
emp(Fn, Ln,_, degree(_,_,school(ScN, StN), Yr))
    & ivy(ScN, StN) & Yr>1979 & Yr<1990
% FACTS
% the emp relation of Figure 4 and the
% Old Ivy Leauge binary relation of Figure 8
```

Figure 16. A sample database.

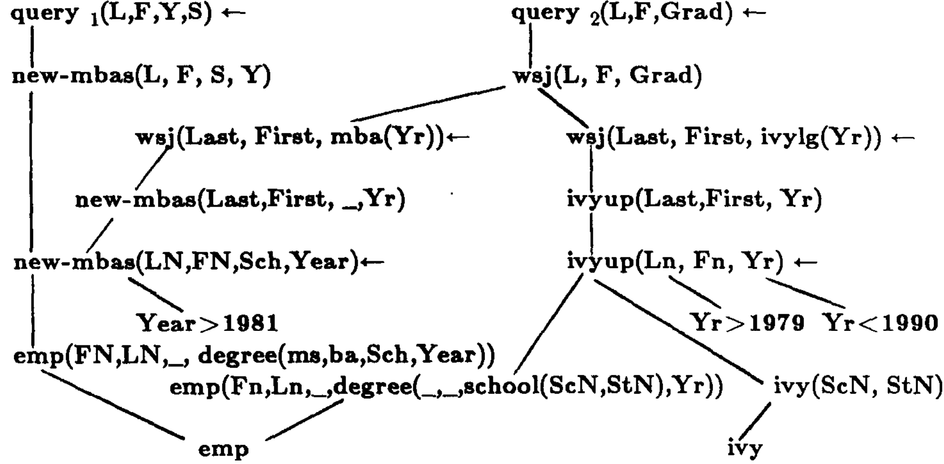

Figure 17. The predicate connection graph for Example 5.1 (Figure 16).

6. Compilation of NRS Formulas

Each rule, say

$$Q \leftarrow P_1 \& \cdots \& P_m \qquad (6.1)$$

undergoes four transformation steps. For notational simplicity, we can assume that the comparison predicates are all collected at the end of the rule (no generality is lost here since the order of predicates is immaterial).

The first step, called the γ-transformation, replaces (6.1) by following two rules:

$$Q \leftarrow Q^{\gamma}(X_1, \cdots, X_r) \qquad (6.2)$$

$$Q^{\gamma}(X_1, \cdots, X_r) \leftarrow P_1 \& \cdots \& P_m \qquad (6.3)$$

where $X_1, \cdots, X_r$ denote the variables of Q *listed in the very order in which they first appear in the tail of (6.1).* We will refer to a rule such as (6.2) as a γ-rule. This rule reorders and (possibly) combine the variables of the *simple* predicate which constitutes the tail into the (possibly) complex head predicate.

The second step, called $\sigma\pi$-transformation, factors out the comparison predicates of (6.3), if any of these are present, and projects out any tail variable not appearing in the head. Since comparison predicates are at the end of the tail, they can be denoted by $P_{k+1},\ldots,P_m$, with $1<k<m$. Then (6.3) is replaced by the pair of rules:

$$Q^{\gamma}(X_1, \cdots, X_r) \leftarrow$$
$$Q^{\sigma\pi}(Y_1, \cdots, Y_n) \& P_{k+1} \& \cdots \& P_m \qquad (6.4)$$

$$Q^{\sigma\pi}(Y_1, \cdots, Y_n) \leftarrow P_1 \& \cdots \& P_k \qquad (6.5)$$

Where $Y_1,\ldots,Y_n$ is the list of variables in the tail of (6.3) in the order of their first appearance (thus, $X_1, \cdots, X_r$ is a sublist of $Y_1,\ldots,Y_n$). A rule such as (6.4) will be called a $\sigma\pi$-rule.

The third step transforms each ESP predicate in the tail of (6.5). An ESP predicate is defined as one where some argument is not a variable (i.e., such an argument is a constant or a complex term). Say that P_j is an ESP predicate in the tail of (6.5). Then P_j is replaced by $P^{\rho}{}_j(Z_1, \cdots, Z_{h_j})$ with $Z_1, \cdots, Z_{h_j}$ denoting the non-anonymous variables of P_j. Thus rule (6.5) is replaced by:

$$Q^{\sigma\pi}(Y_1, \cdots, Y_n) \leftarrow P'_1 \& \cdots \& P'_k \qquad (6.6)$$

where P'_j stands for P_j if this is not an ESP predicate, and for $P^{\rho}{}_j(Z_1, \cdots, Z_{h_j})$, defined above, otherwise.

A rule such a (6.6), consisting of a simple predicate, with only variables as arguments, and where the list of the head variables is identical to that of the tail variables, will be called a J-rule.

Finally, we have the ρ-transformation step. For each $P^{\rho}{}_j$ predicate defined above, we have to add

$$P^{\rho}{}_j(Z_1, \cdots, Z_{h_j}) \leftarrow P_j \qquad (6.7)$$

A rule such as (6.7), where the variables in the tail's complex predicate appear as simple variables in the head, will be called a ρ-rule.

In summary, because of these transformations, the original rule (6.1) was then replaced by the following rules:

1) A γ-rule (6.2),
2) a $\sigma\pi$-rule (6.4),
3) a J-rule (6.5),
4) and a bunch of ρ-rules, such as (6.7).

The complete transformation will therefore be called a γ–$\sigma\pi$–J–ρ mapping.

Example 6.1: The γ–$\sigma\pi$–J–ρ mapping applied to the rule of Figure 8 yields:

ivyup(Ln, Fn, Yr) ←   ivyup$^{\gamma}$ (Fn, Ln, Yr)

ivyup$^{\gamma}$ (Fn, Ln, Yr) ←
ivyup$^{\sigma\pi}$ (Fn, Ln, Yr) &
Yr > 1979 & Yr < 1990

ivyup$^{\sigma\pi}$ (Fn, Ln, Yr) ←
emp$^{\rho}$(Fn, Ln, ScN, StN, Yr) &
ivy(ScN, StN)

emp$^{\rho}$(Fn,Ln,ScN,StN,Yr)←
emp(Fn,Ln,_,degree(_,_,school(ScN,StN),Yr))

Figure 18. The result of the γ-$\sigma\pi$-J-ρ mapping on the rule of Figure 8.

Definition 6.1: A set of queries, rules and facts is said to be equivalent to another, when for each query in the first set there exists one in the second set that produces exactly the same result, and vice-versa.

Then we have the following Lemmas:

Lemma 6.1: Given an arbitrary set of facts, rules, and queries on these, the γ-$\sigma\pi$-J-ρ transformation on rules produces an equivalent set.

Proof: This property holds for all four steps, hence for their composition.

A rule defines a mapping from sets of m-tuples to sets of n-tuples, where m denotes the sum of the argument counts for the predicates in the tail, and n is the argument count for the head. An ERA operator (or operator expression) that defines the same mapping will be said to implement the rule.

Lemma 6.2: Every γ-rule can be implemented by a combine operator.

Proof: In a γ-rule, such as (6.2) the simple variables in the tail all appear, possibly reordered and combined, in the head.

Lemma 6.3: Every $\sigma\pi$-rule can be implemented by an ESP operator.

Proof: In a $\sigma\pi$-rule, the simple variables in the $Q^{\sigma\pi}$ predicate in (6.4) — the Y's may be subject to some selection conditions; moreover, some of them may not be present in the head. This mapping can be implemented by a selection followed by a projection, which, in turn, can be replaced by an ESP operation.

Lemma 6.4: Every J-rule can be implemented by a Join operator.

Proof: In a rule such as (6.6) all the arguments in the tail are simple and they are all contained, without reordering, in the head. Such a rule can then be implemented by a sequence of (equi)join operators as per the usual Domain Calculus to Relational Algebra mapping [Ullman 80].

Lemma 6.5: Every ρ-rule can be implemented by an ESP operator.

Proof: We have now a rule such as the last rule in Figure 18, where there is a single complex predicate on the right, having as arguments complex terms or constants, and a head having as simple arguments all variables in the tail, with no change in their order. This transformation can be implemented by the ESP operator constructed by applying the following recursive procedure to the tail predicate and its subterms.

Translation of ρ-Rules into ESPs: For each argument j of the term, do:

(1) if j denotes a constant, say C, add the condition $j=C$ to the σ list,

(2) if j denotes a non-anonymous variable, add j to the π-list,

(3) if j denotes an anonymous variable followed by a right parenthesis — the last argument in the term— then add the corresponding arity condition,

(4) if j denotes a complex term with functor f, then add the condition $j.0=f$ to the σ-list and recursively apply this procedure to its subarguments.

It is then easy to show, by induction on the depth of the tree representing the term, that this transformation yields a ρ operator that contains all conditions expressed by the ρ rule, thus it implements the rule itself.

We can connect in a tree the various operators implementing the γ-$\sigma\pi$-J-ρ subrules. This tree, will be called the γ-$\sigma\pi$-J-ρ tree, for the rule. It thus follows that:

Lemma 6.6: Every rule can be implemented by a γ-$\sigma\pi$-J-ρ expression.

Example 6.1: The γ-$\sigma\pi$-J-ρ tree for the rule of Figure 7 — i.e., the rules of Figure 18 — is given in Figure 15.

Example 6.2: The γ-$\sigma\pi$-J-ρ tree for the rule of Figure 6 is given in Figure 14. Observe that we have omitted a trivial (unary join) operator.

Example 6.3: The translation of the first **wsj** rule in Figure 16, yields an operator $\rho[count=4/1,2,3]$ followed by a $\gamma[1,2,mba(3)]$; the ESP and join produced by the γ-$\sigma\pi$-J-ρ mapping define trivial identities and have been omitted. The arity condition $count=4$ is, strictly speaking, unnecessary, since the arity of these tuples is already guaranteed by the context; our policy is to include the arity condition anyway, to ensure a more direct correspondence to the original predicate connection graph, and to the filter-based implementation discussed in the next section. The second **wsj** rule in Figure 16, map into one non-trivial operator: $\gamma[1,2,ivylg(3)]$.

ERA DAG for an NRS Database: This graph is constructed from the predicate connection graph as follows:

Rule nodes: For each such a node, the comparison predicates hanging from the node are first eliminated and then the node itself is replaced by the γ–$\sigma\pi$–J–ρ tree implementing the rule.

Goal nodes: Each such a node is replaced by an union operator (that reduces to the identity operator when the node has only one child).

The ERA DAG for Example 5.1 is given in Figure 20.

We can thus state the following theorem:

Theorem 6.1: Queries and rules in a NRS Database can be implemented by ERA expressions.

Proof: Every rule or query corresponds to a node in the ERA DAG. Thus, we only need to extract the subgraph hanging from that node, and then by node-splitting transform it into a tree, which represents the desired ERA expression.

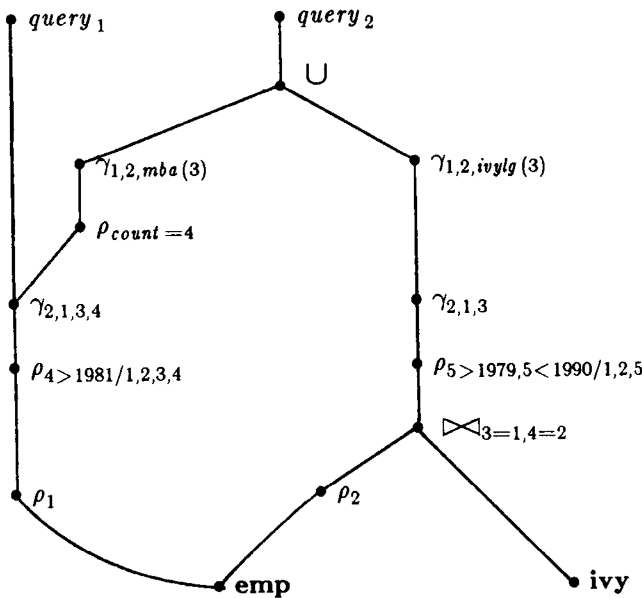

Figure 20. ERA DAG for Example 5.1 (Fig. 16), with:

$$\rho_1 = \rho_{4.0=degree\ ,4.1=ms\ ,4.2=ba\ /1,2,4.3,4.4}$$
$$\rho_2 = \rho_{4.0=degree\ ,4.3.0=school\ /1,2,4.3.1,4.3.2,4.4}$$

7. Implementation

ERA DAGs and trees lend themselves to efficient implementations via well-established database techniques for query optimization and support by special-purpose hardware.

ERA graph improvements represent the first obvious step towards efficient implementation. All trivial nodes can be eliminated and identical operators can be merged, as it was done in Figure 14. Moreover, combine and ESP operators should be migrated downwards when possible, to cut down on the cardinality of intermediate results and the number of components in each tuple. The applicability of these transformations is limited by join nodes, and also by multiple parent nodes when conflicting conditions are passed down from the ancestors. A solution consists in splitting multiple parent nodes, and supporting different queries by different subtrees defining specialized subexpressions.

The Rule/Goal Tree approach proposed in [Ullman 85] can be viewed as a radical application of the specialized subexpression approach. Such a tree differs from a predicate connection graph inasmuch as there is only one query node, and all unification bindings are propagated across the whole tree. Rule/Goal Trees can also be transformed into equivalent ERA expressions using the mapping described in the last section. As it will be discussed in future reports, the end result is equivalent to that obtained by starting from the ERA DAG, passing unary operators down and splitting the multiple ancestor nodes. Since both the ERA DAG and the ERA tree approach offer certain advantages and drawbacks, it is expected that designers may want to use one or the other, or intermediate solutions, according to the situation at hand.

The new ERA operations introduced can be supported with existing technology. A very useful device that has emerged from database technology is the *filter* [Bancilhon 80]. Filters are capable of performing projections and selections "on-the-fly", and they have been used as the building block of various database machines.[2]

In their more advanced realizations filters are implemented as Finite-State Automata. Similar techniques can be used to realize the ESP and combine operators of ERA. For instance the finite state analog of the operator $\rho[4.0{=}degree\ ,4.1{=}ms\ ,4.2{=}ba\ /1,2,4.3,4.4]$ of Figure 15 is given in Figure 21.

The mapping from an ESP operator a to the finite-state equivalent is simple. In addition to **start, end, pass** and **fail**, we need states for each element in the σ/π lists, plus their siblings, their ancestors, and the siblings of their ancestors (e.g, if 4.3 is present then we also need 4.1, 4.2, 4.4 and 1, 2 and 3). Then we connect these states in the natural order (3 after 2 and 4.1 after 3), and we derive the input/output patterns from the σ/π lists.

The finite-state filter operates as follows. Tuples are processed as they stream along, with the help of a

[2] Possible duplicates resulting from projections can not be eliminated on-the-fly. If users do not want duplicates, then union operators will also have to eliminate duplicates. Thus, duplicate elimination can simply be obtained by leaving unary union operators in our ERA DAG.

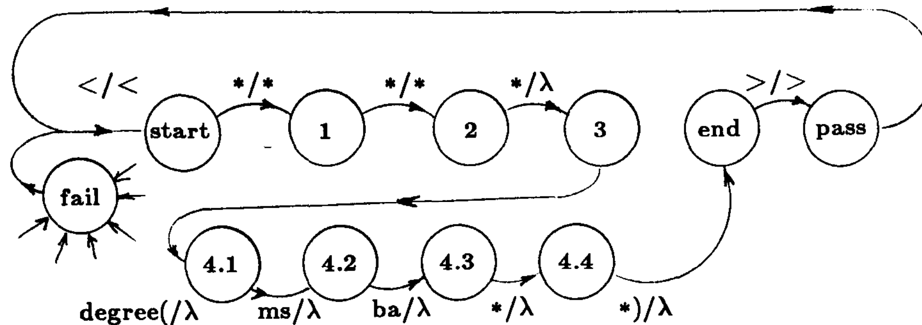

Figure 21. The finite state machine for ρ[4.0=*degree* ,4.1=*ms* ,4.2=*ba* /1,2,4.3,4.4]

temporary buffer. If a tuple fails some input pattern test, the machine is set to the **fail** state and the buffer is disposed of. Otherwise, the **pass** state is eventually reached and the tuple is accepted. In either case the process resumes with the next tuple.

Actually, our filters are more powerful than the regular finite state automata, since they can process complex fields as indivisible units. For instance, state 4.3 must be able to extract and return the term **school(usc,ca)** from the last fact in Figure 4, disregarding the comma that, normally, represents a token separator, but here must be ignored because it appears inside parentheses. A simple mechanism to implement this policy is to use a parenthesis counter.

As discussed in future reports, simple devices are also available to implement the combine operators.

8. Conclusion

In this paper we have taken a first, but significant, step towards efficient implementations of Horn Clause Logic using database techniques. We have studied the unification problem, which is at the core of every LP system, and provided a simple extension of relational algebra that implements it efficiently. This result may be of surprise to readers who are familiar with the work presented in [Dwork 84], on the computational complexity of unification. In our approach however, we process Horn Clauses that normally require full unification in two stages. Full unification is only used for the first stage — the compilation stage— where either the predicate connection graph, or the rule/goal tree, is built. During the second stage, i.e., at execution time, all that is left to do is *matching,* which is defined as the unification of two terms where only one contains variables. Now, since matching is amenable to parallel execution [Dwork 84], our results are are in agreement with those described in [Dwork 84].

The approach presented in this paper proposes a new framework for exploiting the and/or parallelism that is implicit in pure Horn Clauses. The task of supporting Horn Clauses via one-way unification (matching) is performed by operators of two kinds. The first kind of operators include ESP and Combine operators. There is little to be gained by using parallelism to implement these operators, since they operate "on-the-fly", i.e., the process is dominated by the rate of flow of data from storage. Operators of the second kind, which include unions and joins, offer great opportunities for speedup through parallelism. These are the sort of problems with which all implementors of DBMS and database machines are well-acquainted. The techniques used range from careful selection of join sequences and join algorithms (query optimization) to the use of parallel hardware.

This is only a preliminary result and much work remains to be done. We have been successful in reducing the implementation problem for Logic to a database implementation problem for non-recursive safe Horn Clauses, but not for the general case. Our frame example of Figures 9 and 10, for instance, cannot be compiled with our technique since the third rule in Figure 10 is not safe. To handle this rule we need to allow arithmetic expressions and recognize as safe, variables that are derived from other safe variables via calculations. Also one needs to follow more closely the order of bindings of variables to values, as with the capture rules described in [Ullman 85]. Compilation techniques for this case are given in [Zaniolo 85]. Finally, we need to investigate how the ERA operators can be used in conjunction with the recursive rule compilation techniques proposed by various authors [Chang 81, Ullman 85].

Acknowledgments

I am grateful to Charles Kellogg, Roger Nasr, Rolf Stachowitz and the referees for many helpful comments on an early draft of this manuscript, and to Francois Bancilhon and Dick Tsur for stimulating discussions.

References

[Aho 79] A. Aho and J. Ullman, "Universality of Data Retrieval Languages", *Proceedings of 6th POPL*, 1979.

[Bancilhon 80] F. Bancilhon and M. School, "On Designing an I/O Processor for a Relational Database Machine", *Proc. ACM-SIGMOD on Management of Data*, pp. 93-93g, 1984.

[Campbell 84] J. A. Campbell, "*Implementations of Prolog*," Ellis Horwood Ltd., 1984.

[Chandra 82] A. Chandra and D. Harel, "Structure and Complexity of Relational Queries", *JCSS 25, 99-128 (1982)*

[Chang 81] C. Chang, "On evaluation of Queries Containing Derived Relations in Relational Database" *In Advances in Data Base Theory*, Vol. 1, H. Gallaire, J. Minker and J.M. Nicolas (eds.), Plenum Press, New York, 235-260, 1981.

[Codd 72] Codd E. F., "*Relational Completeness of Data Base Sublanguages*," Courant Computer Science Symp., Prentice Hall, 1972.

[Dwork 84] C. Dwork, P. Kanellakis and J. Mitchell, "*On the Sequential Nature of Unification*", Journal of Logic Programming, Vol. 1, pp 35-50, 1984.

[Gallaire 84] H. Gallaire, J. Minker and J.-M. Nicolas, "Logic and Data Bases: A Deductive Approach", *Computing Surveys, Vol. 16, No 2, June 1984*.

[Henschen 84] L. Henschen and S. Naqvi, "On Compiling Queries in Recursive First-Order Data Bases", *JACM*, Vol 31, pp. 47-85, January 1984.

[Jarke 84] Jarke M., J. Clifford and Y. Vassiliou, "An Optimizing Prolog Front-end to a Relational Query," *Proc. ACM SIGMOD Conference on Management of Data*, 1984.

[Kowalski 75] R. Kowalski, "A Proof Procedure Using Connection Graphs", *JACM*, 31:1, pp. 4785, 1975.

[Kowalski 84] R. Kowalski, personal communication, April 1984.

[McKay 81] D. McKay and S. Shapiro, "Using Active Connection Graphs for Reasoning with Recursive Rules", *Proceedings 7th IJCAI*, pp. 368-374, 1981.

[Naqvi 83] S. Naqvi and L. Henschen, "Synthesizing Least Fixed Point Queries into Non-recursive Iterative Programs", *Proceedings IJCAI 83*, 1983.

[Li 84] Li D., "*A Prolog Database System*," Research Institute Press, Letchworth, Hertfordshire, England, 1984.

[Parker 85] D. S. Parker et al. "Logic Programming and Databases" *Proc. First International Workshop on Expert Database Systems*, Kiawah Island, S.C., Oct. 1984.

[Reiter 78] R. Reiter, "Deductive Question Answering on Relational Data Base", In *Logic and Data Bases*, H. Gallaire and J. Minker (eds.), Plenum Press, New York, pp. 149-177, 1978.

[Smith 77] Smith, J.M. and D.C.P. Smith, "Database Abstractions: Aggregation and Generalization," *ACM Trans. Database Systems*, 2, 2, pp. 105-133, 1977.

[Tsur 84] Tsur S. and C. Zaniolo, "On the Implementation of GEM: supporting a semantic data model on a relational back-end," *Proc. ACM-SIGMOD Conference on Management of Data*, 1984.

[Ullman 80] Ullman, J., "*Principles of Database Systems*," Computer Science Press, 1980.

[Ullman 85] J. Ullman, "Implementation of Logical Query Languages for Databases" *Proc. ACM-SIGMOD Conference on Management of Data*, 1985.

[Winston 79] P. H. Winston, "*Artificial Intelligence*," Addison Wesley, 1979.

[Zaniolo 83] C. Zaniolo, "The Database Language GEM", *Proc. ACM-SIGMOD Conference on Management of Data*, 1983.

[Zaniolo 84] Zaniolo C., "Object-Oriented Programming in Prolog," *Proc. Int. Logic Programming Symposium*, IEEE, 1984.

[Zaniolo 85] Zaniolo C., "The Compilation of Horn Clauses into Extended Relational Algebra," manuscript in preparation.

[Zloof 85] Zloof M.M., "Query by Example," *Proc. AFIPS NCC 44*, pp. 431-438, 1975.