New Generation Computing, 4 (1986) 273-285
OHMSHA, LTD. and Springer-Verlag

The Alexander Method — A Technique for The Processing of Recursive Axioms in Deductive Databases*

J. ROHMER, R. LESCOEUR and J. M. KERISIT
Centre de Recherche Bull
68, Route de Versailles, 78430 Louveciennes, France.

Received 29 July 1985
Revised manuscript received 26 June 1986

Abstract We propose a technique for handling recursive axioms in deductive databases. More precisely, we solve the following problem:

Given a relational query including virtual relations defined from axioms (Horn clauses, with variables in the conclusion predefined in the hypotheses), which can be recursive, how to translate this query into a **relational program**, i. e. a set of relational operations concerning only real relations (not virtual). Our solution has the following properties:

- the program to evaluate the query always terminates,
- the relational program is produced by a pure compilation of a source query and of the axioms, and is independent of the data values (there is no run-time),
- the relational operations are optimized: they **focus** towards the computation of the query, without needless computations.

As far as we know, the Alexander Method is the first solution exhibiting all these properties.
This work is partly funded by Esprit Project 112 (KIMS).

Keywords: Recursion, Logic, Deductive Database, Saturation

§1 Introduction

Given a relational query including virtual relations defined from axioms (Horn clauses, with variables in the conclusion predefined in the hypotheses), which can be recursive, how to translate this query into a **relational program**, i. e. a set of relational operations concerning only real (not virtual) relations.

* This work is partly funded by Esprit Project 112 (KIMS).

This problem is central in the research on deductive databases[9] and more generally in the domain of the relationship between logic and database.[4]

The objective is to avoid, given a very large deductive database, to just store data as Prolog clauses, since one knows that the backtrack policy of Prolog performs the operations one tuple at a time, with the worst case for join operations, i. e. nested loops.

The idea is to take advantage of the optimization techniques developed for database management systems. We want to use these techniques, and only these techniques, to access data, instead of backtracking. Numerous authors have addressed these problems, but no complete solutions has been published at present time.

In Ref. 2), one distinguishes two choices for non-recursive axioms:

- "enumeration", consisting of expanding the query down to basic (real) relational expressions.
- the other one, consisting of creating a sequence of calls to the database.

These two approaches are not able to deal with recursive axioms since they yield either infinite expressions or infinite calls.

Several authors give partial solutions for some subsets of recursions.[3,5] Other authors solve the whole problem,[9,10,12] but at the price of the generation of a lot of useless facts, due for instance to a "forward chaining" strategy, without focusing on the very query.

A rigorous formalization of the problem, and a partial solution can be found in Ref. 8).

The solution detailed in this paper is as following:

(1) The initial query is first considered as a goal "ã la Prolog", to be solved in **backward chaining**.
(2) This query is **translated** into a set of clauses to be processed in **forward chaining**.
(3) This last set of forward clauses is itself **translated** into **pure relational operations**.

This last transformation (point 3) was studied earlier in our group in the framework of an inference engine: BOUM,[10] which uses what we call a Delta-Driven Mechanism, to *saturate* in forward chaining a set of Horn clauses. BOUM compiles strictly (statically) a set of Horn clauses into a relational program which cyclically saturates the set of conclusions. Automatic elimination of duplicates ensures **termination** of the saturation.[9] Some examples of this transformation are given in annex to this paper.

The translation from backward chaining into forward chaining (point 2) is the subject of this paper, and we call it the "ALEXANDER METHOD". Why this name?

The problem of recursive axioms in deductive database is in some sort the Gordian knot of databases. For instance, the naive and native method to deal with recursion, i. e. backward chaining with backtrack and several stacks, really looks like a knot.

On the contrary, forward chaining exhibits many interesting properties of **simplicity**. But forward chaining has the drawback of computing **all** the possible answers to all possible queries, without focusing on the minimal calculus to compute the answer to **one** particular query.

For instance if we want to know the ancestors of someone, it is useless to compute first **all** the ancestors of everybody, then to retain just the ancestors of this person.

The idea will be, for a given query, to compute a new set of clauses, which, when processed in forward chaining, will produce only the useful information. In some sense, we will use forward chaining to **simulate** backward chaining.

To cut the Gordian knot, we shall cut a recursive goal into two pieces:

- one problem
- solutions.

For instance, the goal Ancestor(Jean, x) will be cut into:
— a literal like Pb_Ancestor(Jean) which can be interpreted as "The problem of finding the successors of Jean exists",
— literals like Sol_Ancestor(Jean, Louis) which can be interpreted as "Louis is a solution to the problem Pb_Ancestor(Jean)".

To go from backward chaining to forward chaining, we need clauses which will handle Pb_Ancestor and Sol_Ancestor literals.

Examples:

- Pb_Ancestor(x), Q → R
  "**if** there is the problem of finding the successors of x, and if Q is true, **then** ..."
- A → Sol_Ancestor(x, y)
  "**if** A is true, **then** y is a solution"

With these intuitive ideas in mind, let us process an example by hand: the goal Ancestor(Jean, x).

The axioms are:

(1) Father(x, y) → Ancestor(x, y)
(2) Father(x, y), Ancestor(y, z) → Ancestor(x, z)

The rule (1) gives:

(1.1) Pb_Ancestor(x), Father(x, y) → Sol_Ancestor(x, y)
"If there is the problem of finding the successors of x, and if x is the father of y, then a solution is y."

The rule (2) gives:

(2.1) Pb_Ancestor(x), Father(x, y), Ancestor(y, z) → Sol_Ancestor(x, z)
"If there is the problem of finding the successors of x, and if x is the father of y, and if y is an ancestor of z, then a solution is z."

But this rule contains itself a goal Ancestor(y, z), thus it must itself be transformed. This goal will itself be cut into two pieces, yielding two new rules: (2.2) and (2.3).

(2.2) Pb_Ancestor(x), Father(x, y) → Pb_Ancestor($\bar{y}$)
"If there is the problem of finding the successor of x, and if x is the father of y, then there exists the problem of finding the successor of y".

This rule generates new Pb_Ancestor, which, through for instance rule 1.1), will generate new Sol_Ancestor.

(2.3) Sol_Ancestor(y, z) → Sol_Ancestor(x, z)

which expresses that the solutions to the "y" problem are also solutions to the "x" problem.

In fact, this rule (2.3) is not correct, since x appears in conclusion and not in hypotheses. Thus, we must **transmit** information x between rules (2.2) and (2.3). For that purpose, we create a new predicate named **context** or **continuation**.

The final version of (2.2) and (2.3) is now:

(2.2′) Pb_Ancestor(x), Father(x, y) → Pb_Ancestor(y), Cont(y, x)
(2.3′) Sol_Ancestor(y, z), Cont(y, x) → Sol_Ancestor(x, z)

In a second step, as explained in Annex 1, such forward rules can be translated into pure relational programs.

§2 The Transformation Algorithm

Let E be a set of Horn clauses such that variables in the conclusion also appear as variables in the hypotheses.

The names of the literals are partitioned into two classes:

- the ones to be evaluated in forward chaining (FCL)
- the ones to be evaluated in backward chaining (BCL)

We want to **saturate** in this set of clauses **only** the literals belonging to FCL.

Example:

The set E consists of:

P(x) Ancestor(x, y) → OK(y)
Father(x, y) → Ancestor(x, y)
Father(x, y) Ancestor(y, z) → Ancestor(x, z)
FCL = P, OK, Father

BCL = Ancestor

We want to find all the OK literals, without finding all the Ancestors, but only the ones needed for the computation of the OK literals.

Note: This is a superset of the problem of querying a deductive database. Example,

Consider the query: find all the successors of Jean?
We associate to this query a new clause Ancestor(Jean, x) $\rightarrow$ Answer(x), and we declare that the literal Ancestor belongs to BCL and Answer to FCL.

The detail of the transformation algorithm follows:

Notations:

X, X1, Y, Y1, Z, Z1, W represent sequences of literals in clauses.
A and B represent individual literals.

The transformation process concerns a couple of clauses:
— the first one, noted as C1, uses a BCL literal in hypothesis, i. e. is of the form:

C1 = X A Y $\rightarrow$ Z   where A belongs to BCL

— the second one, noted as C2, has a BCL literal as a conclusion, i. e. is of the form:

C2 = W $\rightarrow$ B   where B belongs to BCL

The transformation happens if and only if literals A and B are unifiable. If s is the corresponding substitution, let

Goal = s(A) = s(B)

Let g be a symbol coding Goal in a unique manner (a number or a string for instance). This coding is explained in the next paragraph.
From C1 and C2, we first build two rules, by applying the substitution s to them:

s(C1) = X1 Goal Y1 $\rightarrow$ Z1
and s(C2) = W1 $\rightarrow$ Goal

We cut s(C1) into two new rules:

C1_left_g = X1 $\rightarrow$ Pb_g(iv) Cont_C1_g(iv, cv)
C1_right_g = Sol_g(iv, ov) Cont_C1_g(iv, cv) Y1 $\rightarrow$ Z1

This needs some explanation of the notations.
"C1_left_g" represents a symbol built by replacing C1 and g by their string value.

A third new rule is obtained from s(C2):

C2_def_g = Pb_g(iv) W1 → Sol_g(iv, ov)

The meanings of iv, ov, cv are the following:

iv: set of the variables common to X1 and Goal (input variables)
ov: set of the variables common to Goal and (Y1 or Z1) and not in iv (output variables)
cv: set of the variables common to X1 and (Y1 or Z1) but not belonging to Goal (continuation variables)

These new three rules (C1_left_g, C1_right_g, C2_def_g) are added to the set E.

Then a new transformation process is attempted, by choosing another couple of rules C1 and C2 in E.

The process terminates when no new rules can be generated.

Proofs of termination, correctness and completeness of the transformation have been given in Ref 6).

§3 Computation of the Unique Name of a Goal

During the transformation between:

X A Y → Z
and W → B,

we must give a unique name g to the unifier of A and B.

Moreover, we must characterize the goal according to the input and output status of its variables. For instance:

— in P(x) Ancestor(x, y) ... → ...
the goal is to compute the successors of x
— in P(x, y) Ancestor(x, y) ... → ...
the goal is to check if x is the ancestor of y

In the first case, the name of the problem will be Ancestor.1.0. In the second case, it will be Ancestor.1.1.

We put a "1" for the input parameters and a "0" for the output parameters.

If a literal has n arguments, it can generate up to 2 power n goal names.

One can differentiate the goals not only by the input/output mode of the parameters, but also by the presence and value of constants.

Example:

R1 = X(y) A("max", x, y) → Z(x, y)
R2 = W(a, b) → A(a, b, "joe")

Unification gives:

a = "max", x = b, y = "joe"

and the rules after substitution are:

X("joe") A("max", x, "joe") $\rightarrow$ Z(x, "joe")
W("max", b) $\rightarrow$ A("max", b, "joe")

We can incorporate the constant values into the name of the goal, which becomes, with evident notations:

A. *max*. 0. *joe*

Here iv (input variables) is empty and ov (output variables) corresponds to x (and b).

Consequently, the final transformed rules are:

X("joe") $\rightarrow$ Pb_A. *max*. 0. *joe*   Cont_R1_A. *max*. 0. *joe*
Sol_A. *max*. 0. *joe*(x)   Cont_R1_A. *max*. 0. *joe* $\rightarrow$ Z(x, "joe")
Pb_A. *max*. 0. *joe*   W("max", b) $\rightarrow$ Sol_A. *max*. 0. *joe*(b)

Here, we have performed a precompilation of the unification, by propagating the constants into the rules, and by making them implicit in the predicate names.

Note: This strategy may not always be interesting. We could just consider that constants in hypotheses are input variables, and constants in conclusions are output variables.

§4 A Detailed Example of Transformation

Let us consider our standard example:

RA = P(a)   A(a, b) $\rightarrow$ OK(b)
RB = F(x, y) $\rightarrow$ A(x, y)
RC = F(x, y)   A(y, z) $\rightarrow$ A(x, z)

We shall follow the steps and notations of Section 2.

First step

C1 = RA = P(a)   A(a, b) $\rightarrow$ OK(b)
C2 = RB = F(x, y) $\rightarrow$ A(x, y)

The substitution s is (x = a, y = b)

Goal = A(a, b) = A(x, y)

iv = a = x
ov = b = y
cv = empty

The goal name g is A.1.0

Transformed rules:

RA_left_A.1.0 = P(a) → Pb_A.1.0(a) Cont_RA_A.1.0(a)
RA_right_A.1.0 = Sol_A.1.0(a, b) Cont_RA_A.1.0(a) → OK(b)
RB_def_A.1.0 = Pb_A.1.0(x) F(x, y) → Sol_A.1.0(x, y)

Second step
Another transformation is possible between rules RA and RC.

C1 = RA = P(a) A(a, b) → OK(b)
C2 = RC = F(x, y) A(y, z) → A(x, z)

Substitution s = (a = x, b = z)

Goal name g = A. 1.0
It is the same goal name as in the first step, thus rule RA needs not to be transformed again. But rule RC is transformed, giving:

RC_def_A.1.0 = Pb_A.1.0(x) F(x, y) A(y, z) → Sol_A.1.0(x, z)

This rule contains A in hypothesis, and must be cut ! To shorten the names, let us give it the name RD in place of RC_def_A.1.0 (in practice, all these names are internally coded, as in the examples shown in Annex 2).

This RD rule must be transformed with rules RB and RC, as what happened with RA in the first step. In both cases, the goal name g is still A.1.0, thus RB and RC need not be transformed again. But rules RD has to be transformed:

iv = y ov = z cv = x

RD_left_A.1.0 = Pb_A.1.0(x) F(x, y) → PB_A.1.0(y) Cont_RD_A.1.0(y, x)
RD_right_A.1.0 = Sol_A.1.0(y, z) Cont_RD_A.1.0(y, x) → Sol_A.1.0(x, z)

Finally, we have replaced the set of clauses (RA, RB, RC) by a set of five forward rules:

RA_left_A.1.0
RA_right_A.1.0
RB_def_A.1.0
RD_left_A.1.0
RD_right_A.1.0

§5 Conclusion

We think that the Alexander Method propose the first complete solution to transform **by a strict compilation process** recursive axioms into pure relational algebra operations.

Since the submission of the paper, other methods have appeared (Refs. 1, 7, 13)) which deal with the same problem; as far as we know, only the last one has been implemented.

As for any compiling techniques, there are many opportunities for local and global optimization of source and object code. This work is now under progress.

The interest of the Alexander Method is to transform a complex problem into simpler ones (forward chaining, relational algebra). It allows to add new facilities to existing software packages **without modifying** them: adding backward chaining to an inference engine with deductive facilities, and adding recursive axioms to classical DBMS ...

References

1) Bancilhon, Maier, Sagiv, Ullman, "Magic Sets and other strange ways to implement Logic Programming," MCC Technical Report number DB 121-85, Oct., 1985.
2) Chakravarthy, U. S., Minker J. and Tran D., "Interfacing predicate logic languages and relational, " Proc. lst Int. Conf. on Logic Programming, Marseille, 1982.
3) Chang, "Deduce 2. Further investigations of deduction in Relational Databases," in Ref. 4), pp. 201-236.
4) Gallaire, H. and Minker, J. (eds.), *Logic and Databases*, Plenum Press, New York, 1978.
5) Henschien, L. J. and Naqui, S. A., "On compiling queries in Recursive first order Databases, " *CACM*, Jan., 1984.
6) Kerisit, "Preuve de la Methode d'Alexander par une approche algébrique," BULL rapport interne, May 1986.
7) Lozinskii, "Evaluating queries in Deductive Databases by generating," IJCAI, 1985.
8) Marque-Pucheux, G., "Interfacing Prolog and Relational Data Base Management System," *ICOD-2*, Sept., 1983.
9) Nicolas, J. M. and Yazdanian, K., "An outline of B.D.G.E.N.: a deductive DBMS, " *Information Processing*, IFIP, 1983.
10) Pugin, J. M. and Jamier, R., "Le Moteur d'Inférence BOUM," *Rapport de DEA*, Ecole Centrale de Paris, Juin, 1983.
11) Reiter, R., "Deductive Question-Answering on relational Databases," in *Logic and Databases*, Plenum Press, pp 149-177, 1978.
12) Shapiro, J. D., *Principles of Database Systems — 2nd Edition*, Computer Science Press, 1982.
13) Vieille, L.,"Recursive Axioms in Deductive Databases. The Query-Subquery approach," Expert Database System Conference, Charleston, Apr. 1986.

ANNEX 1

Transformation of a set of forward rules into relational operations: Delta driven mechanism

Saturation of a set of rules is processed as follows:

· **INIT**

— for each rule R: P1, ..., Pn $\rightarrow$ C,
one computes: ΔCR = P1 join P2 ... join Pn

```
    — for each literal C in conclusion of at least 1 rule,
      one computes $\Delta$C = Union of all $\Delta$CR
                              where C is a conclusion of R

• ENCORE
    — for each rule  R: P1, ..., Pn → C,
      one computes:
        _CR1 = P1 join P2 ... join Pn
           :    :    :    :
        _CRi = P1 join ... join $\Delta$Pi ... join Pn
           :    :    :    :
        _CRn = P1 join P2 ... join $\Delta$Pn
    — then_CR = Union of $\Delta$CRi
    — for each literal C in conclusion of at least 1 rule,
      one computes:
        $\Delta$C = Union of all $\Delta$CR
          where C is a conclusion of R
        $\Delta$C = C Difference $\Delta$C
          C = C Union $\Delta$C

• TEST
    if not all $\Delta$C are empty goto ENCORE else FIN.
```

ANNEX 2

Examples of the Alexander Transformation on some different rule bases

1) "Ancestor" defined as follows:

Father(x, y) → Ancestor(x, y)
Father(x, y), Ancestor(y, z) → Ancestor(x, z)

with a goal rule:

Ques(x), Ancestor(x, y) → Ancestor-of-ques(x, y)

After transformations (seen in fourth part of the article), the rules to be saturated are:

r1: Ques(x) = Pb_Ancestor.1.0(x), Cont_0(x)
r2: Sol_Ancestor.1.0(x, y), Cont_0(x) → Ancestor-of-ques(x, y)
r3: Pb_Ancestor.1.0(x), Father(x, y) → Sol_Ancestor.1.0(x, y)
r4: Pb_Ancestor.1.0(x), Father(x, z) → Pb_Ancestor.1.0(z), Cont_1(z, x)
r5: Sol_Ancestor.1.0(y, z), Cont_1(y, x) → Sol_Ancestor.1.0(x, z)

(See in annex 3, an example of saturation of this rule Base on a Facts Base)

2) "Ancestor" defined as follows:

Father(x, y) → Ancestor(x, y)
Ancestor(x, y), Ancestor(y, z) → Ancestor(x, z)

with the same goal rule as in 1).
After transformations, the rules will be:

Ques(x) → Pb_Ancestor.1.0(x), Cont_0(x)
Sol_Ancestor.1.0(x, y), Cont_0(x) → Ancestor-of-ques(x, y)
Pb_Ancestor.1.0(x), Father(x, y) → Sol_Ancestor.1.0(x, y)
Pb_Ancestor.1.0(x) → Pb_Ancestor.1.0(x), Cont_1(x)
Sol_Ancestor.1.0(x, y), Cont_1(x) → Pb_Ancestor.1.0(y), Cont_2(y, x)
Sol_Ancestor.1.0(y, z), Cont_2(z, x) → Sol_Ancestor.1.0(x, z)

After some evident simplifications on **Cont 0 and Cont 1**, the rule Base to be saturated will be:

r1: Ques(x) → Pb_Ancestor.1.0(x)
r2: Sol _Ancestor.1.0(x, y), Pb_Ancestor.1.0(x) → Ancestor-of-ques(x, y)
r3: Pb_Ancestor.1.0(x), Father(x, y) → Sol_Ancestor.1.0(x, y)
r4: Sol_Ancestor.1.0(x, y), Pb_Ancestor.1.0(x) →
Pb_Ancestor.1.0(y), Cont_2(y, x)
r5: Sol_Ancestor.1.0(y, z), Cont_2(z, x) → Sol_Ancestor.1.0(x, z)

(NB: **r2** and **r4** have the same hypotheses and can be replaced by:

r6: Sol_Ancestor.1.0(x, y), Pb_Ancestor.1.0(x)→
Ancestor-of-ques(x, y), Pb_Ancestor.1.0(y), Cont_2(y, x)

but it's not the purpose of this article to develop such optimizations.

3) "Family Friends" defined as:

Friend(x, y) → Ancfr(x, y)
Father(x, y), Ancfr(y, z) → Ancfr(x, z)

with the goal rule:

Ancfr(*leo*, y) → Fam-friend(y)

where *leo* is a constant.

The transformed rule Base will be:

r1: → Pb_Ancfr.1.0(*leo*)
r2: Sol_Ancfr.1.0(*leo*, y) → Fam-friend(y)
r3: Pb_Ancfr.1.0(x), Friend(x, y) → Sol_Ancfr.1.0(x, y)
r4: Pb_Ancfr.1.0(x), Father(x, y) → Pb_Ancfr.1.0(y), Cont_0(y, x)
r5: Sol_Ancfr.1.0(y, z), Cont_0(y, x) → Sol_Ancfr.1.0(x, z)

ANNEX 3

Execution of example of 1 in annex 2.

The Facts base is:

Father(remi, lisa)
Father(remi, julie)
Father(jeanic, remi)
Father(paul, jeanic)
Father(charles, paul)
Father(oto, rita)

Father(rita, julie)

Ques (paul)
Ques (oto)

From 1), the executable Rules, with an obvious renumbering, will be:

```
r1: Ques(x) → Pb_Ancestor.1.0(x)
r2: Sol_Ancestor.1.0(x, y), Ques → Ancestor-of-ques(x, y)
r3: Pb_Ancestor.1.0(x), Father(x, y) →
            Sol_Ancestor.1.0(x, y), Pb_Ancestor.1.0(y), Cont_1(y, x)
r4: Sol_Ancestor.1.0(y, z), Cont_1(y, x) → Sol_Ancestor.1.0(x, z)
```

Delta-saturation on these rules gives the following intermediate results, cycle after cycle:

```
Cycle 1:
    r1:
        Pb_Ancestor.1.0 : (oto), (paul)

Cycle 2:
    r3:
        Sol_Ancestor.1.0 : (paul jeanic), (oto rita)
        Pb_Ancestor.1.0  : (jeanic), (rita)
        Cont_1           : (jeanic paul), (rita oto)

Cycle 3:
    r2:
        Ancestor-of-ques : (paul jeanic), (oto rita)
    r3:
        Sol_Ancestor.1.0 : (jeanic remi), (rita julie)
        Pb_Ancestor.1.0  : (remi), (julie)
        Cont_1           : (remi jeanic), (julie rita)

Cycle 4:
    r3:
        Sol_Ancestor.1.0 : (remi lisa), (remi julie)
        Pb_Ancestor.1.0  : (lisa)
        Cont_1           : (lisa remi), (julie remi)
    r4:
        Sol_Ancrstor.1.0 : (paul remi), (oto julie)

Cycle 5:
    r2:
        Ancestor-of-ques : (paul remi), (oto julie)
    r4:
        Sol_Ancestor.1.0 : (jeanic lisa) (jeanic julie)

Cycle 6:
    r4:
        Sol_Ancestor.1.0 : (paul lisa), (paul julie)

Cycle 7:
    r2:
            Ancestor-of-ques : (paul lisa), (paul julie)
Cycle  8:
            Nothing New To Be Produced ...
END.
```