THE SUPER-MAGIC IMPLEMENTATION METHOD FOR DATALOG PROGRAMS

by

Eran Grey

Technical Report #452

March 1987
THE SUPER-MAGIC IMPLEMENTATION METHOD FOR DATALOG PROGRAMS

Eran Gery

Computer Science Department
Technion - Israel Institute of Technology
Haifa 32000, Israel

ABSTRACT

We present a method which transforms a set of pure Datalog rules and a query into another set of rules which can be evaluated more efficiently in a naive bottom up computation. The method is an extension of the idea of Magic-Sets [B+86] applicable to the general case of Datalog rules.

1. INTRODUCTION

Given a relation defined recursively by horn clauses, it can be generated by a bottom up computation of the least fixed point of a relational transformation (forward chaining). In this paper we consider the case where not the whole relation is required, i.e. we have bindings for some of the relation attributes. Obviously we can compute the whole relation and then select only the required tuples from it, but since we are not interested in the entire relation, there is a strong motivation to compute a cover set that contains the required tuples, with as small cardinality as possible.

Aho and Ullman [AU 79] proposed a solution for a simple class of linear recursive rules, which simply propagate the initial bindings into the rules, and the resulting relation contains exactly the required tuples. This optimization strategy is efficient to its applicability domain, but this domain extends only to a special sub-class of the linear rules.

The Magic sets method [B+86] is a strategy which transform the clauses into a new rules system, that can compute a sub-relation which is a superset of the required tuples, by adding "guard" predicates into the original rules, restricting them to generate only "useful tuples", i.e. tuples which belong to the answer set, or tuples which are useful as intermediate results to deduce answer-tuples. The method also creates rules which define these "guards", the so called Magic-sets. The optimized system evaluates the given query in two phases.
The first phase computes the Magic predicates (we use the terms "predicate" and a relation symbol interchangeably); since they can be computed independently of the relation, while the second phase actually evaluates the relation which is the super set of the result relation.

The Magic sets method is applicable to the general case of recursion but actually the optimization is effective only for linear-recursive rules and some special cases of non-linear-recursive rules.

Recently the "Alexander" method [RLK 86] introduced a rule optimization method, which is based on a principle similar to the Magic-sets method, but it may be effective for the general case of recursion. The disadvantage of the Alexander method is its dependency on the order of the predicates in the rule’s body, i.e. there are cases in which the same rule with a different order of its body predicates yields a much more efficient optimized rule’s system. An additional advantage of the Alexander method relative to Magic-sets is an optimization which saves intermediate common results for usage in rules which have common computation in their body. In other words, it performs “factoring” of rules with a common sub-body thereby avoiding redundant computations which occurs in the Magic-sets computation.

In this paper we generalize the Magic sets method to the general case of recursion. The main idea is relaxing the restriction of the original method to compute the Magic predicates independently of the result relation, and letting the Magic rules to access the result relation predicate. This way we get a single phase computation where the Magic predicates and the virtual relation predicates are mutually incremented. The method also defines some “good” order in the rule bodies. Thus, it is independent of the original order.

The suggested method builds another set of rules, of the bottom-up evaluation of which results in the answers to the given query. Given a set of rules, the task of the bottom up evaluation itself can also be performed more efficiently than by the naive way, but this issue is outside the scope of this paper.

The rest of the paper is organized as follows:

In section 2 we present a motivating example of the transformation using a typical non-linear recursive query, and prove the correctness of the transformation for this case.

In section 3 we present the algorithm for the general case, and its correctness proof, and finally we also offer another optimization based on the idea of the Alexander method, avoiding redundant computations of the modified rules system. Section 4 ends with a discussion.
2. A MOTIVATING EXAMPLE

We define the binary relation $P$ (using prolog notation) as follows:

(i) $P(x, y) :- A(x, x 1), B(x 2, x 3), C(x 3, x 4)$

(ii) $P(x, y) :- D(x, y)$

and the query is:

$Q(x) :- P(a, x)$

Here $A, B, C, D$ are base relations, and $P$ is a virtual (or defined) relation.

2.1 Transforming the rules into an adorned system

The main idea in restricting the intermediate relations is building the guard predicates, which will hold only values that would have been bound to a rule argument during a top-down evaluation process. Thus, although the computation that we actually perform is bottom-up, we shall simulate the bindings propagation during a top-down process. Therefore, we informally justify some of the transformation steps in terms of a top-down computation, although the program is actually evaluated bottom-up. The adorned rule system [U 85] corresponding to a set of rules, is a set of Horn rules in which every virtual (non base) predicate is associated with an adornment. An adornment is a string which associates with each of the predicate's attributes the symbol $b$ or $f$, which means free or bound, respectively. The meaning of an adorned predicate, for example $P^f$ is: whenever $P^f$ occurs in a rule body it will be called with its first argument bound and its second argument free. We can interpret the first argument as an "input parameter" and the second argument as an "output parameter".

We start to elaborate the system from the query rule. Here the query $P(a, x)$ calls $P$ as a goal with its first argument bound to $a$ and the second argument free, so the query induces the adorned predicate $P^f$, and we get the adorned query rule:

$Q(x) :- P^f(a, x)$

The next step is create from every "P rule" (a rule with head $P$) its $P^f$ form. Every new adorned version of a rule may generate new adorned predicates from the rule body. The process terminates when no new adorned predicates are introduced. (For every rule whose head is a predicate with arity $n$ we may get at most $2^n$ versions in the adorned system).

Given an adorned predicate $P^a$ where $a$ stands for some adornment, and a rule $P :- Q$, the adornments
which will be associated with its body predicates \((Q)\) depend on the order in which we are going to schedule them as goals. We order the predicates in the body left to right, where we assume that the predicates are invoked as goals from left to right. The criterion of a "good" scheduling is that every predicate will be called as a goal at least with one bound argument, or in other words, we want to avoid adornments which are all free. In the sequel we will see that this ordering will guarantee that every rule will have a "guard" predicate at least for one of its arguments.

Going back to rule (i) we have to generate its \(P^V\) form. Since the first argument \((x)\) is bound (that is the property of a \(P^V\) rule) we first call to \(A(x, x_1)\), whereby \(x_1\) becomes bound, then we call \(P\) with its second argument bound, so we introduce a new adorned predicate \(P^b\). This call binds \(x_2\), since it is an "output parameter" of the call. In this step we see the main difference between this method and the Magic sets; i.e. the use of results from recursive calls for binding variables. In the successive calls, we refer \(x_2\) as bound variable. Hence, since \(x_2\) is bound we call \(B\) whereby \(x_3\) becomes bound, and the next call to \(P\) will be again of the form \(P^b\).

The resulting rule is:

1. \(P^b(x, y) :- A(x, x_1), P^b(x_2, x_1), B(x_2, x_3), P^b(x_3, x_4), C(x_4, y).\)

Now we create \(P^b\) version to rule (ii).

2. \(P^b(x, y) :- D(x, y).\)

Using the same principle we get the \(P^b\) versions:

3. \(P^b(x, y) :- C(x_4, y), P^b(x_3, x_4), B(x_2, x_3), P^b(x_2, x_1), A(x, x_1).\)

4. \(P^b(x, y) :- D(x, y).\)

Since no more adorned predicates were generated the process terminates and we have a complete adorned system.

2.2 Adding the Magic predicates

For every adorned rule we define a Magic predicate which contains all the bound variables (according the adornment) in the head of the rule.

Example: suppose the rule head is \(P^b(x, y)\). We add to its body an unary predicate which refers to the variable \(x\).

Since the property of the Magic predicate is that it "predicts" all the bindings that would have been
generated in a top down process to the variables to which it refers, adding this predicate to the rule body will restrict the set of tuples that will be generated in the bottom up computation that is actually performed.

We mark \( M.P^\alpha \) the Magic predicate corresponding to a rule with adorned head \( P^\alpha \), so in the example we consider we mark two Magic predicates, \( M.P^W \) and \( M.P^B \), and we modify the adorned system as follows:

1. \( P^W (x, y) :\!-\! M.P^W (x), A(x, x_1), P^B(x_2, x_1), B(x_2, x_3), P^W (x_3, x_4), C(x_4, y). \)
2. \( P^W (x, y) :\!-\! M.P^W (x), D(x, y). \)
3. \( P^B (x, y) :\!-\! M.P^B (x), C(x_4, y), P^B(x_3, x_4), B(x_2, x_3), P^W (x_2, x_1), A(x, x_1). \)
4. \( P^B (x, y) :\!-\! M.P^B (x), D(x, y). \)

Now to complete the transformation we define the rules that computes the Magic predicates.

2.3 Generating the Magic rules.

As we mentioned before the Magic predicates "predict" the set of bindings to some argument of a rule that would be generated in a top-down evaluation. We build the rules defining the Magic predicates according to this principle. First, we define the rules for \( M.P^W \). To do this we have to look for all the rules which call \( P^W \), and extract the subrule which evaluates the variable which is the actual binding in some specific goal-call.

The first call to \( P^W \) is in the given query, and the actual binding is simply the constant \( a \). That yields the first Magic rule which is simply

\[
5. M.P^W (a).
\]

The next call to \( P^W \) is in rule 1, and the actual variable which holds the bindings is \( x_3 \), so we extract from the rule the part which computes \( x_3 \) and we get:

\[
6. M.P^W (x_3) :\!-\! M.P^W (x), A(x, x_1), P^B(x_2, x_1), B(x_2, x_3).
\]

From rule 3 we build one more rule for \( M.P^B \):

\[
7. M.P^B (x_2) :\!-\! M.P^B (y), C(x_4, y), P^B(x_3, x_4), B(x_3, x_2).
\]

According to the same idea, we create the rules for \( M.P^B \):

\[
8. M.P^B (x_1) :\!-\! M.P^W (x), A(x, x_1).
\]

\[
9. M.P^B (x_4) :\!-\! M.P^W (y), C(x_4, y).
\]

Rules 1 to 9 plus the the modified query are the new system, for which we perform bottom up iteration to find its least fixed point. The required answer is the final value of the relation \( Q \).
2.4 Correctness proof of the transformation

First we state the correctness criterion of the transformation:

(*) \( \forall y (P(a,y) \Rightarrow P(a,y)) \)

Where \( P \) stands for the relation defined by the original Datalog program, and \( P' \) is the relation defined by the transformed program. This criterion originates from the fact that in the modified system the answers are derived from \( P' \) instead of \( P \). To prove this statement we prove the following two claims which imply the statement (*):

(*)1) \( \forall x \mu (M.P_{M}^{\mu}(x) \Rightarrow \{P(x,z) \Rightarrow \mu_{\mu}(x,z)) \)

(*)2) \( \forall y,x (M.P_{M}^{\mu}(y) \Rightarrow \{P(x,y) \Rightarrow \mu_{\mu}(x,y)) \)

Lemma 1

\( \forall x,y (P_{\mu}(x,y) \Rightarrow P_{\mu}(x,y)) \)

\( \forall x,y (P_{\mu}(x,y) \Rightarrow P_{\mu}(x,y)) \)

Actually this lemma implies one side of the equivalence in each of the claims (*1),(*2), which means that the relations \( P_{\mu},P_{M} \) are contained in the original relation \( P \). This follows directly from the fact that the Magic predicates are only restricting guards, and can not contribute any extra tuples to the computation.

Note the independence of \( x \) from \( M.P_{M}^{\mu} \) and of \( y \) from \( M.P_{M}^{\mu} \).

Lemma 2

(*)1) \( \forall x,z (M.P_{M}^{\mu}(x) \Rightarrow \{P(x,z) \Rightarrow \mu_{\mu}(x,z)) \)

(*)2) \( \forall y,z (M.P_{M}^{\mu}(y) \Rightarrow \{P(x,y) \Rightarrow \mu_{\mu}(x,y)) \)

This lemma is a kind of completeness result, showing that all relevant tuples are indeed generated by the adorned system. Here the claim is indeed made only for magic values \( x \) and \( y \).

Definition: The depth of a tuple \( P(x,y) \) is the minimum number of iterations in the fixpoint computation of the in the original system, to derive the tuple.

We prove the claims by induction on the tuple depth.

Base: tuple depth is one.

If the tuple was generated after one iteration, it must have been deduced from the non-recursive rule (rule ii). If \( M.P_{M}^{\mu}(x) \) holds, then the same tuple will be generated also in the modified system by rule 2. If \( M.P_{M}^{\mu}(y) \) holds, then the same tuple is generated by rule 4.
The induction hypothesis:

For any tuple \( P(x,y) \) whose depth is less than \( n \), the claims (**1'**),(**2'** ) hold.

Induction step: Suppose \( M.P^M(x) \) holds for some tuple \( P(x,y) \) (otherwise the claim holds trivially). Since the tuple depth is not one, it was generated by rule (i):

\[
P(x, y) := A(x, x_1), P(x_2, x_1), B(x_2, x_3), P(x_3, x_4), C(x_4, y).
\]

The tuples \( P(x_2, x_1), P(x_3, x_4) \) have depth smaller than \( n \). Using the Magic rule 8, and the fact that \( M.P^M(x) \) holds, we get that \( M.P^B(x) \) holds, and using the induction hypothesis we conclude that \( P^M(x_1, x_2) \) holds. From Magic rule 6 and the previous facts, we conclude that \( M.P^M(x_3) \) will be generated, and from the induction hypothesis again that \( P^M(x_3, x_4) \) holds. Now using rule 1, we obtain that \( P^M(x,y) \) holds.

If \( M.P^B(y) \) holds, then using similar arguments we prove that \( P^B(x, y) \) is obtained.

Theorem:

The modified program is equivalent to the original one.

Lemma 1 and Lemma 2 imply claim (i), and for \( x = a \) we get

\[
\forall y \ (M.P^M(a) \Rightarrow P(a,y) \Rightarrow P^M(a,y)).
\]

Using the fact that \( M.P^M(a) \) holds by definition, we conclude that the correctness criterion (*) holds.

2.5 Making the bottom up evaluation more efficient.

A prominent fact is that the transformation process creates rules with common body, since every rule generates Magic rules which have common body with itself. For example, rule 1 generates the two Magic rules 6 and 8:

1. \( P^M(x, y) := M.P^M(x), A(x, x_1), P^B(x_2, x_1), B(x_2, x_3), P^M(x_3, x_4), C(x_4, y) \)

6. \( M.P^M(x_2) := M.P^M(x), A(x, x_1), P^B(x_2, x_1), B(x_2, x_3) \)

8. \( M.P^B(x_1) := M.P^B(x), A(x, x_1) \)

We can modify these three rules into another equivalent set of rules, which discard the redundant computations by keeping intermediate relations. This idea is very similar to the decomposition algorithm used by the "Alexander" method, which makes this method more efficient than the original "Magic sets" method even in its domain of applicability (the linear-recursive rules). We introduce two unary intermediate relations, \( T_1,T_2 \), and build the following rules (we use the ";" sign to denote two rules with the same body):
3. DESCRIPTION OF THE GENERAL TRANSFORMATION

In this section we describe the transformation for the general case of Datalog programs. We assume a set of range restricted [BR 86] Horn-rules without constant symbols and evaluable predicates, and a query of the form \( Q(\bar{y}) : P(\bar{a}, \bar{y}) \) where \( \bar{a} \) stands for vector of constant symbols, and \( \bar{y} \) stands for the "output" variables. The differences of the general case from the typical example considered above are:

1. Relations are not necessarily binary.
2. The placement of recursive calls in the body is arbitrary.
3. There may be some mutually recursive defined relations.
4. The number of recursive calls in the rules body is not limited.

3.1 Transforming the system into an ordered adorned system

First, we define a procedure to generate an adorned rule from non-adorned rule and some adornment:

Given an adornment \( \alpha \) and a rule \( r \), with head \( P \) (we assume that \( \alpha \) is suitable to the arity of \( P \) ), we define the transformation of \( r \) into its \( \alpha \)-adorned form. We generate an adorned rule \( r' \) by ordering the goals in \( r \) and adding adornments to the virtual goals, which depends on this ordering. We build the body of \( r' \) according to the order from left to right.

We define for each body goal an attribute \( \text{num} \), which is the place of the goal in the new rule. For the virtual predicates we define one more attribute, \( \text{adornment} \), which is their adornment. Initially, all the attributes have undefined values. We define a set of bound variables \( \text{bound} \), which initially contains the head variables which are marked bound by \( \alpha \). When we "schedule" a goal, we add all its variables to \( \text{bound} \).

Remark: in the following procedure the term "bound variable" means a variable which is in \( \text{bound} \), "not numbered goal" means a goal which its \( \text{num} \) attribute is not yet defined.
i := 1;
WHILE (there exist body goals which are not numbered) DO
    IF (exist not numbered ground goal A which contains a bound variable) THEN
        A.num := i;
        bound := bound \cup \{ the variables of A \}.
    ELSE IF (exist not numbered virtual goal Q which contains bound variable)
        Q.adornment := mark all bound variables "b" others "f";
        Q.num := i; bound := bound \cup \{ the variables of Q \}.
    ELSE IF (exist not numbered ground goal) THEN
        A.num := i;
        bound := bound \cup \{ the variables of A \}.
    ELSE
        select any not numbered virtual goal Q;
        Q.adornment := mark all variables "f";
        Q.num := i;
        bound := bound \cup \{ the variables of Q \}.
    END IF
    i := i + 1
END WHILE.

Now we generate \( \gamma' \) by binding the head predicate the adornment \( \alpha \) and order its body goals according to their numbering left to right, and the virtual goals are assigned their adornments. Note that the induced ordering is not unique, depending on the selection by the existential quantifiers.

Now, using the previous procedure we complete the transformation of the system into an adorned system.

Let Rules be the set of the original rules (excluding the given query). New is the set of the modified rules (actually the output), initially empty. Adorned is a set of adorned predicates (with the corresponding adornments), in containing the adorned predicate from the query rule. Now we define the following procedure:

FOR all \( P \in \text{Adorned} \) DO
    FOR all \( r \in \text{Rules} \) such that \( r \) is unifiable with \( P \) DO
        apply the adornment procedure on \( r \) with the adornment of \( P \)
        where \( r' \) is the output rule.
        Adorned := Adorned \cup \{ all the adorned predicates of \( r' \) \}.
        New := New \cup r'
    END FOR
END FOR

Remark: The termination of this procedure is guaranteed since the cardinality of \( \text{Adorned} \) is bound.

3.2 Adding the magic predicates

First we add to every adorned rule a magic predicate. For each rule \( r' \) with head \( P^\alpha(x, y) \) where \( P \) stands for some arbitrary predicate symbol, \( \alpha \) stands for some adornment, \( x \) stands for the arguments of \( P \)
which are bound in the adornment, and \( \bar{y} \) stands for the free variables, we add to the rule:

\[
\mathfrak{r} : P^\alpha(\bar{x}, \bar{y}) \leftarrow \bar{W}.
\]

the magic predicate \( M.P^\alpha(\bar{y}) \) and we get:

\[
\mathfrak{r} : P^\alpha(\bar{x}, \bar{y}) \leftarrow M.P^\alpha(\bar{y}), \bar{W}.
\]

Remark: if \( \alpha \) is all free, we will not add a magic predicate; since we have no information on any of its arguments. In the discussion, we say more about the "all free" possibilities.

Now, for every adorned predicate in the body of \( \mathfrak{r} \) we build a magic rule (except for the all-free predicates).

Let \( R^\beta(\bar{u}, \bar{v}) \) be a predicate on the body of \( \mathfrak{r}' \), where \( \bar{u} \) stands for the arguments which are bound in \( \beta \):

\[
\mathfrak{r}' : P^\alpha(\bar{x}, \bar{y}) \leftarrow \bar{S}, R^\beta(\bar{u}, \bar{v}), \bar{T}.
\]

Then we build the magic rule:

\[
M.R^\beta(\bar{u}) \leftarrow \bar{S}.
\]

The previous procedure should be applied on every rule in the adorned system. Finally, we add the query from the adorned system.

3.3 A proof outline for the general case

The claim which constitutes the transformation correctness is an extension of the claim discussed in the special case in section 2. We assume that the virtual predicates numbered \( P_i \), \( i=1,n \).

The general claim comprises the following set of claims, for each adorned predicate:

\[
\forall \bar{x}, \forall \bar{y} (M.P^\alpha_{\beta}(\bar{x}) \Rightarrow (P_i(\bar{x}, \bar{y}) \Leftrightarrow P^\alpha_{\beta}(\bar{x}, \bar{y})))
\]

Where \( \bar{x} \) are the variables which are bound in \( \alpha \). We note that \( M.P^\alpha_{\beta} \) may be empty if \( \alpha \) binds no variables, and the claim in these cases holds trivially. Actually, one side of the equivalence of each claim, that \( P^\alpha_{\beta}(\bar{x}, \bar{y}) \Rightarrow P_i(\bar{x}, \bar{y}) \) is trivial (see section 2), and we prove the other side by induction on each tuple depth, as in section 2. We give a generic proof for each of the claims in the set.

Base: \( P_i(\bar{x}, \bar{y}) \) is a tuple of depth one.

Then it must be generated from a nonrecursive rule of the form:

\[
P_i(\bar{x}, \bar{y}) \leftarrow A(\bar{w}).
\]

Where \( A \) stands for some ground relation, or conjunction of base relations. Then we have an adorned rule of the form:
then the claim follows.
Suppose the set of claims holds for each tuple whose depth less than \( d \). Let \( P_i(\vec{x}, \vec{y}) \) be a tuple whose depth is \( d \), and \( MP_i(\vec{x}) \) holds. (Otherwise the claim holds trivially.) Then this tuple was generated by a rule of the form:

\[
P_i(\vec{x}, \vec{y}) := Z_1, Q_1(x_1, y_1), ..., Z_n, QA(\vec{x}_m, \vec{y}_m), Z_{n+1}.
\]

from the facts \( Q_i(\vec{x}_i, \vec{y}_i) \), and some other base facts, where \( Z_i \) stands for base predicates. Then in the modified system we have a rule of the form:

\[
P_i(\vec{x}, \vec{y}) := MP_i(\vec{x}), Z_1, Q_1(\vec{x}_1, \vec{y}_1), ..., Z_n, QA(\vec{x}_m, \vec{y}_m), Z_{n+1}.
\]

Now we show that every of the required facts \( Q_i(\vec{x}_i, \vec{y}_i) \) which are of depth less than \( d \), are computed in the modified system. We shall prove it by induction on \( i \). The base is \( Q_i(\vec{x}_1, \vec{y}_1) \). According the transformation algorithm, we have a magic rule of the form:

\[
M.Q_i(\vec{x}_1, \vec{y}_1) := MP_i(\vec{x}), Z_1.
\]

This magic rule implies that \( M.Q_i(\vec{x}_1, \vec{y}_1) \) holds, and using the induction hypothesis (of the outer induction) \( Q_i(\vec{x}_1, \vec{y}_1) \) holds. Now we assume that \( Q_i(\vec{x}_i, \vec{y}_i) \) holds for \( i < n \). We prove that \( Q_a(\vec{x}_m, \vec{y}_m) \) holds.

According the transformation algorithm, we have the magic rule:

\[
M.Q_a(\vec{x}_m, \vec{y}_m) := MP_i(\vec{x}), Z_1, Q_i(\vec{x}_1, \vec{y}_1), ..., Z_n.
\]

Using our inner induction hypothesis we get that \( M.Q_a(\vec{x}_m, \vec{y}_m) \) holds, and using the outer induction hypothesis we prove that \( Q_a(\vec{x}_m, \vec{y}_m) \) holds. The fact that \( Q_i(\vec{x}_i, \vec{y}_i) \) holds for every \( i \), implies that \( P_i(\vec{x}, \vec{y}) \) is generated.

3.4 Optimizing the modified system

As discussed in section 2, the modified system causes redundant computations in its bottom-up evaluation, since the magic rules which are derived from the adorned rules have common sub-bodies. We modify the translation scheme by generating relations which holds intermediate tuples, and using them we discard the redundant computations.

Suppose we have an adorned rule:
1. \( P \leftarrow Z_1, Q_1, Z_2, Q_2, Z_3 \)

Then we generate two magic-rules for \( Q_1, Q_2 \) as follows:

2. \( M.Q_1(\ldots) \leftarrow Z_1 \)

3. \( M.Q_2(\ldots) \leftarrow Z_1, Q_1, Z_2 \)

Since rule 2's body is a prefix of rule 3's body, and rule 3's body is a prefix of rule 1's body, the idea is to evaluate rule 3 using intermediate results from rule 2, and so on. To implement this idea we introduce two new predicates \( T_1, T_2 \) and modify the previous 3 rules as follows:

First we save the required results from rule 2, using \( T_1 \):

\[ T_1(\bar{x}) ; M.Q_1(\ldots) \leftarrow Z_1 \]

Where \( \bar{x} \) are the common variables between \( Z_1 \) and the other body predicates.

Now, using \( T_1 \) we define rule 3, and \( T_2 \) which is required to define rule 1:

\[ T_2(\bar{y}) ; M.Q_2(\ldots) \leftarrow T_1, Q_2, Z_2 \]

Where \( \bar{y} \) stands for the common variables between the last body predicates and the P rule body predicates.

Finally we define new rule for \( P \) based on the intermediate relation \( T_2 \):

\[ P \leftarrow T_2, Q_2, Z_3 \]

The equivalence between the two rules sets is quite obvious. Formally, it can be proved by induction on the tuples depth. Obviously, the equivalence is on the common relations, i.e. excluding the temporary relations.

4. DISCUSSION

In this section we present some properties of the method, which may give some estimate on its efficiency. As discussed in [B+ 86], it is unreasonable to expect proofs of optimality of such optimization methods. As mentioned there, even in methods to optimize regular relational algebra operations we do not find any optimal results but optimization rules which are justified by "folk wisdom", like "pushing selections inside".

Here, we also use a strategy like "pushing selections inside", which is actually pushing the bindings from the query into the bottom up computations. Intuitively, a "Magic sets oriented" method has to support the following two goals:

---

Technion - Computer Science Department - Technical Report CS0452 - 1987
1. Define Magic predicates for as many rules as possible.
2. Generate Magic rules which compute Magic predicates as small as possible.

It is quite obvious that these two goals are necessary to restrict the computation size.

We define a class of rules so that we can state some properties of the suggested method, when applied to rules belonging to this class. However, the method is applicable and may be efficient also in other cases.

Definition: *Well-connected rule.*

Two predicates \( A, B \) (in a body of some rule) are connected if:

1. They have a common variable, or:
2. There exist (in the body of the rule) an intermediate predicate \( T \) to which both \( A \) and \( B \) are connected.

A well-connected rule is a rule in which every two body predicates are connected.

In conventional database terms it means that the rule body implies a natural join expression without any Cartesian product.

Definition: *Well-connected program.*

A well-connected program is a rule system in which every rule is well-connected.

Next we state two properties of the transformation when applied to well-connected programs.

Theorem:

If the original program is well-connected then the modified program has the following two properties:

1. The modified rules are also well-connected (including the Magic rules!).
2. There are no all-free adorned predicates, i.e. all the modified rules are guarded by Magic predicates.

These properties follow from the steps of the transformation algorithm, and can be proved by induction on these steps.

The importance of the first property is that the generated Magic sets is of "reasonable" size, not "huge" Magic sets which actually do not restrict the computation.

The second property ensures that all the generated rules are potentially restricted. A drawback of the original Magic sets method is that it does not satisfy the second property. For example, in the program we
present in section 2, it will generate a $P^W$ rule, which means that the method fails to restrict the computation.

The following example illustrates a Magic transformation which does not satisfy the first property:

$$P(x, y) \leftarrow E(x, y)$$

$$P(x, y) \leftarrow A(x, x_1), B(x_2, y), P(x_1, x_2).$$

$$Q(y) \leftarrow P(a, y).$$

In the following transformation, the adorned rules are created without reordering their bodies, hence we bind the goals arguments according to the original left to right calling sequence (however the transformation is correct):

$$P^W(x, y) \leftarrow MP^W(x), A(x, x_1), B(x_2, y), P^{bb}(x_1, x_2).$$

$$P^{bb}(x, y) \leftarrow MP^{bb}(x, y), A(x, x_1), B(x_2, y), P^{bb}(x_1, x_2).$$

The Magic rules:

$$MP^W(a).$$

$$MP^{bb}(x_1, x_2) \leftarrow MP^W(x), A(x, x_1), B(x_2, y).\text{ (*)}$$

$$MP^{bb}(x_1, x_2) \leftarrow MP^{bb}(x, y), A(x, x_1), B(x_2, y).$$

The rule (*) is not well-connected, hence it generates Magic tuples by Cartesian product between two projections of the base relations $A$ and $B$, which results in all the potential domain of bindings to the second argument of $P$. Thus, the second argument of these Magic tuples is useless, and it only enlarges the Magic set. If we keep the connectivity in the Magic rules, such Cartesian products would have been avoided. Therefore, the suggested method in this case will generate only Magic set to the first argument, i.e $MP^W$.

This example contributes some intuition as to why well-connected Magic rules are a "good property".

ACKNOWLEDGEMENT

I would like to thank Nissim Francez for reading and commenting this paper.

REFERENCES


