Optimizing Existential Datalog Queries[†]

Raghu Ramakrishnan

Catriel Beeri

Ravi Krishnamurthy

Computer Sciences Technical Report #773

June 1988

[†]This paper has appeared in Proc. PODS, 1988.

PROC. PODS, 1988

Optimizing Existential Datalog Queries

Raghu Ramakrishnan (1)
Catriel Beeri (2,3)
Ravi Krishnamurthy (2)

ABSTRACT

The problem of pushing projections in recursive rules has received little attention. The objective of this paper is to motivate this problem and present some (partial) solutions. We consider programs with function-free rules, also known as *Datalog* programs. After formally defining existential subqueries, we present a syntactic criterion for detecting them and then consider optimization in three areas: 1) We identify the existential subqueries and make them explicit by rewriting the rules. This, in effect, automatically captures some aspects of Prolog's *cut* operator that are appropriate to the bottom-up model of computation. 2) We eliminate argument positions in recursive rules by "pushing projections". 3) We observe that "pushing projections" in rules also has the effect of making some rules (even recursive rules) redundant and try to (identify and) discard them.

1. Introduction

The problem of efficiently evaluating Horn Clause logic programs has recently received much attention in the database community. The approach has generally been to assume bottom-up fixpoint evaluation as the computation strategy, and then optimize it. It is well known that bottom-up fixpoint evaluation is *complete* [Lloyd 84]. (That is, it produces all facts for the query predicate in the least Herbrand model of the program, which is taken to be the intended semantics.) However, it does not restrict computation by utilizing information in the query, and often computes several irrelevant facts. This problem has been the focus of much work in this area†.

In the context of traditional database query optimization, the problem of utilizing information in the query (in particular, constants) is essentially the problem of pushing selections. Surprisingly, the complementary problem of pushing projections in recursive rules has received little attention. The objective of this paper is to motivate this problem and present some (partial) solutions. We consider programs with function-free rules, also known as *Datalog* programs. After formally defining existential subqueries, we present a syntactic criterion for detecting them and then propose optimization in three areas: 1) We identify the existential subqueries and make them explicit by rewriting the rules. This, in effect, automatically captures some aspects of Prolog's cut operator that are appropriate to the bottom-up model of computation. 2) We eliminate argument positions in recursive rules by "pushing projections". This elimination process is shown to be undecidable. However, a sufficient condition for eliminating arguments is presented. 3) We observe

(1) University of Wisconsin-Madison, work partly done while visiting MCC; (2) MCC, Austin; (3) The Hebrew University of Jerusalem, Israel. This work was supported by the USA-ISRAEL Binational Science Foundation, Grant # 85-00082/2.

† See [Bancilhon and Ramakrishnan 86, 87] for a survey.

that "pushing projections" in rules has the effect of making some rules (even recursive rules) redundant and propose to discard them. It is known that elimination of rules is in general undecidable. A notion of *uniform query equivalence* is proposed and sufficient conditions for discarding rules are presented. In summary, we present a framework for pushing projections in recursive rules and identify some important undecidable problems as well as identifying problems for further research in improving this optimization process.

1.1. Basic Definitions and Notation

A *rule* in a Datalog program has the form:

$$p_0(\bar{X}_0) \text{ :- } p_1(\bar{X}_1), \cdots, p_n(\bar{X}_n)$$

where each p_i is a predicate name, and each $\bar{X}_i$ is a vector of variables or constants. A *predicate occurrence* is a predicate name followed by a list of arguments (i.e., a variable or a constant), in the *body* of a rule. We use upper case letters for variable names, lower case letters for predicate names, and numerals for constants. A *query* is a rule without a head, written as $q(\cdots)$? A finite set of rules, $IDB = \{r_1, ..., r_n\}$, each defining a *derived* predicate, is called the intentional database. The extensional database *EDB* is a finite set of finite *base* relations. Without loss of generality, we will assume that *IDB* contains no facts - all facts are part of the extensional database *EDB*. For a given query q let us denote a *program* P as a triple (Q, *EDB*, *IDB*), and let DB = EDB $\cup$ IDB.

Let a *binary chain program* (defined for a query of the form q(X,Y)?), in which the rules in IDB are of the form:

$$p(X,Y) \text{ :- } q1(X,Z1), q2(Z1,Z2), \ldots, qn(Zn-1,Y)$$

As is well known, there is a *context-free grammar* corresponding to an IDB that is defined by simply dropping the arguments. For example, the above binary chain rule becomes a production in a context-free grammar:

$$P \text{ --> } Q1, Q2, \ldots, Qn$$

The predicates in the IDB correspond to non-terminal symbols of the grammar, the predicates in the EDB correspond to terminal symbols, and the query predicate corresponds to the start symbol.

Let us denote by G the grammar obtained by thus transforming a program P containing the query q. The *language* $L(G,q)$† of a grammar G (with q as the start symbol) is defined to be the set of terminal strings that can be generated from the start symbol by applications of the productions. Let us define the *extended language* $L^{ex}(G)$ of a grammar G to be the set of all strings (including those possibly containing non-terminals) generated from the start symbol using the productions of G.

Given a program $P = (Q, IDB, EDB)$ containing the query, $Q = q(\bar{c}, \bar{X})$, the result of applying P to *DB*, which we also refer to as the *answer* for the query Q on *DB*, is the set of bindings to the vector of variables $\bar{X}$ that make the query expression true with respect to the intentional and the extensional database.

We assume the bottom-up model of execution, in which the answer for a query is computed as follows: We start with the database relations, and with empty derived predicates. The values for the derived predicates are computed in stages. At each stage, we add to each derived predicate all the tuples whose membership in it is implied by the program, given the values for the predicates in the previous stage. The sequence of values of the derived predicates is monotonically

† The query used as the start symbol in the grammar is stated redundantly for ease of exposition.

increasing, and its limit is the final values for these predicates. The answer is obtained by applying the appropriate selection to the query's predicate.

For each fact that belongs to the answer (or to any derived predicate), there exists a finite *derivation tree*, that describes how it is derived from base facts using rules of the program. Let $p(c)$ be a fact in the derived predicate p. Then the tree has $p(c)$ at its root, the leaves are base facts, and each internal node is labeled by a fact, and by a rule which generates this fact from the facts labeling its children. A base fact may be viewed as a derivation tree of height one.

1.2. Existential Queries and Projections

Consider the following rule:

```
q(X,Y) :- a(X,Z), q(Z,Y), c(W).
```

Given some tuple in c, i.e. some W such that c(W) holds, the set of tuples in q is independent of the set of tuples in c. Thus, in order to compute q, we need not compute c, beyond determining whether there exists some tuple for c. We call c(W) an *existential* query (to be formally defined later). Further, suppose we are only interested In values for the first argument of q. This is implicit if the query is defined by the following rule:

```
query(X) :- q(X,Y).
```

In the terminology of relational algebra, the second argument of q is projected out in computing the set of answers. If we consider the rule defining q, it is easy to see that not all values in the second argument of q are required. In fact, for a given value in the first argument, all we require is the existence of some tuple in q with that value. This is in effect the operation of ''pushing'' the projection of the second argument of q (in the head of the rule) into the join of predicates in the body. We call the second argument of q an *existential* argument, and use the term *existential query* to refer to any query which contains an existential argument. An existential argument is therefore one that we would like to avoid computing in its entirety. The problem is to identify such arguments and take advantage of them to optimize the computation without altering the set of answers.

To begin with, we present a semantic definition of an existential argument and an existential query. This is used to construct a syntactic criterion, which is used to produce an adorned program. Our approach to optimizing programs containing existential queries is in three phases. First, we identify existential subqueries of an adorned program. Then, we rewrite the rules using a simple transformation whose only role is to make the existential subqueries explicit by creating rules defining them. Even if no further optimization is possible, this allows us to optimize at run time by not computing multiple solutions when we are only interested in the existence of some solution. This, in effect, captures some aspects of Prolog's *cut* operator that are appropriate to the bottom-up model of computation (and is automatically introduced by the compiler)†.

The second phase consists of 'pushing projection', in which some of the existential argument positions are eliminated, improving the performance of the resulting execution.

The third phase in our approach consists of deleting some rules from the adorned program at compile time by taking advantage of the existential nature of subqueries (which is reflected in the adornments).

---

† Note that our objective differs from that motivating Mendelzon's introduction of cuts (using functional dependencies) into Prolog programs [Mendelzon 85]. While both aim at eliminating redundant computation, we are working with a bottom-up execution model, whereas Mendelzon was optimizing Prolog's backtracking.

Finally, the trimmed adorned program can be further transformed using rewriting algorithms such as Magic Sets or Counting. It is observed that these rewritings are orthogonal to the optimizations discussed in this paper.

We consider the above three phases in detail in subsequent sections.

2. Adorning Existential Programs

The definition of an existential argument (and query) uses the notion of program equivalence. Two IDBs, IDB1 and IDB2 are said to be *query equivalent* with respect to a query Q if for all states of the extensional database, EDB, the answer for Q is the same for the two programs (Q,IDB1,EDB) and (Q,IDB2,EDB).

One of the objectives of this paper is to identify the notions of existential arguments and projections in as general terms as possible. We now present a definition of existential arguments which seems to capture the intuition. Unfortunately, detecting existential arguments under this definition turns out to be an undecidable problem.

Given a program P containing the query Q and a rule r1 in which the literal $p(\bar{X},Y)$ occurs in the body of r1, then the argument position corresponding to Y is said to be an *existential argument* if the following holds:

> Let a new rule $p'(\bar{X},Y') :- p(\bar{X},Y)$ be added to the IDB, and let $p(\bar{X},Y)$ be replaced by $p'(\bar{X},Y')$ in r1. Further, let any occurrence of Y in the head predicate be replaced by Y'. Then, the new IDB is query equivalent to the original IDB with respect to the query Q.

For a predicate occurring in the head of the rule, an argument of that predicate is existential if that argument is existential in all occurrences of that predicate in the bodies of the rules of the program.

A query is said to be an *existential query* if that query, when viewed as the body of a rule with no head literal, has an existential argument.

Lemma 2.1: It is undecidable to test whether an argument position in a given literal is existential for a Datalog program. []

We observe the following properties:

1. If a variable Y appears in a body literal and does not appear anywhere else in the rule, except possibly in an existential argument of the head, then the argument position where Y occurs is existential.
2. If two variables Y and Z are existential and if the rule is transformed for Y as described in the definition of an existential argument, then Z remains existential after the transformation.

These observations underlie the following *adornment algorithm* which acts as a sufficient test for identifying existential arguments.

An adornment is a string of *d*'s (don't care or existential arguments) and *n*'s (needed arguments). An adorned version of a predicate is a *query form* for that predicate. This denotes all queries in which all values must be computed (are 'needed') for arguments that correspond to *n*'s, and arguments that correspond to *d*'s are existential. For example, p^{nd}(Y,Z)? denotes that we are interested in all Y values such that there exists some Z value for which p(Y,Z) is true.

We construct an adorned version of the program P^{e_ad}, from program P, by replacing derived predicates of the program by adorned versions, where for some predicates we may obtain several adorned versions.‡ The process starts from the given query whose *n*/*d* adornments determine an

‡ The adorned programs defined here differ from the adorned programs discussed elsewhere [Ullman 86, Bancilhon et al. 86, Beeri and Ramakrishnan 87] in that they identify existential arguments as compared to the bound/free arguments.

adorned version of the query predicate. If p^a is an adorned predicate, then for each rule that has p in its head, we generate an adorned version for the rule as described below and add it to P^{e_ad}. We then mark p^a. Note that the adorned version of a rule may generate additional adorned predicates. The process terminates when no unmarked adorned predicates are left. Termination is guaranteed, since the number of adorned versions of predicates for any given program is finite.

In choosing an adornment for a literal in the body, an argument is existential (d) if the variable in it does not occur anywhere else in the rule, except possibly in an existential argument of the head predicate. All other arguments are adorned as n.

Lemma 2.2: The adornment algorithm adorns an argument with d only if the argument is existential. []

Example 1: Consider the following program:

Original Program

query(X) :- a(X,Y).
a(X,Y) :- p(X,Z), a(Z,Y).
a(X,Y) :- p(X,Y).

Adorned Program

query(X) :- a^{nd}(X,Y).
a^{nd}(X,Y) :- p(X,Z), a^{nd}(Z,Y).
a^{nd}(X,Y) :- p(X,Y). []

For the rest of the paper, we shall assume that the program has been adorned. We observe that the adorned program usually has more rules than the original program, but makes subsequent optimizations possible. The final program will perform at least as well as the original program, and, as we show in later sections, will often perform significantly better.

3. Optimizing Existential Queries

We discuss the three phases of the optimization of existential queries in this section.

3.1. Identifying Connected Components

Using the adorned rules of the program the connected components of the rule bodies are computed. In a given rule, two variables are called *connected* if they have occurrences in the same predicate. This is extended in the obvious way to connection through a chain of variables, where each adjacent pair shares a predicate. Similarly, two predicates (including the head predicate) are connected if they each contain one of a pair of connected variables, with the constraint that in the case of the head predicate, the variable is *not* in an argument that is existential ('d').

Connectivity is an equivalence relation (both on variables and predicates). The set of predicate occurrences in a rule is therefore the union of connected components. One of these contains the rule's head. Other components, if they exist, are actually existential subqueries that are solved independently of any bindings for the rule's head variables.

Let a rule be denoted as follows:

$$h :- C_1, C_2, \ldots, C_n$$

where the C_i denote the connected components in the rule. Each component C_i which does not contain the head predicate h is replaced by a boolean variable B_i, and we add a new rule defining B_i (which has no associated arguments):

B_i :- C_i

This is done for every rule in the given program. The resulting program can be more efficiently executed by the bottom-up execution strategy since it allows us to capture a form of Prolog's cut in this model of execution. This is because a rule defining a boolean variable can be removed from the fixpoint computation once the variable becomes true.

Lemma 3.1: The above transformation preserves query equivalence for any given query, and further, every rule has a single connected component (which contains the head predicate unless the head is boolean). []

The following example illustrates Step 1 and shows why it is useful.

Example 2: Consider the following rules:

p^{nd}(X,U) :- $q1^{nn}$(X,Y), $q2^{nd}$(Y,Z), $q3^{dn}$(U,V), $q4^{n}$(V), $q5^{d}$(W).
$q4^{n}$(X) :- $q6^{n}$(X).

This would be rewritten into the following set of rules:

p^{nd}(X,_) :- $q1^{nn}$(X,Y), $q2^{nd}$(Y,_), B2, B3.
B2 :- $q3^{dn}$(_,V), $q4^{n}$(V).
B3 :- $q5^{d}$(_).
$q4^{n}$(X) :- $q6^{n}$(X).

Clearly, once B2 (or B3) has been shown true, the rule defining it need not be used further. (Note that in the rewritten rules we have replaced existential variables by '_'.) This rewriting makes it easy to take advantage of connected components which are existential queries since such rules can easily be removed from the fixpoint computation (at runtime) after they have succeeded once. Further, if $q4^{n}$ does not appear anywhere else in the program, the rule defining it can also be discarded after B2 is shown true. []

Next, we consider how the adorned program P^{e_ad} can be further optimized by deleting existential arguments.

3.2. Reducing Argument Positions

Consider the adorned program in the Example 1 presented in the previous section. The following program computes the same answers for the query.

Example 3:

query(X) :- a^{nd}(X).
a^{nd}(X) :- p(X,Z), a^{nd}(Z).
a^{nd}(X) :- p(X,Z).

The second argument of a has been projected out because it is an existential argument. This projection has in fact been pushed through the recursion. []

The problem we consider in this section is whether the projection implied by an existential argument can be optimized effectively. One way of doing this is to push the projection as far as possible, including, if possible, through recursion. In some cases, as in the above example, the adornments indicate if this is possible.

An algorithm for commuting selects with the LFP operator was presented in [Aho and Ullman 79], and they observed that the same algorithm could be used to commute projects with the LFP operator. The modified program in Example 3 can also be obtained using their algorithm. It can also be obtained using an algorithm for pushing projections presented in [Kifer and Lozinskii 85],

which is the direct analog of their *static filtering* algorithm for pushing selections [Kifer and Lozinskii 85, Bancilhon and Ramakrishnan 86, 87]. The Kifer and Lozinskii algorithm is, to our knowledge, the only completely specified algorithm for pushing projections in recursive rules.

Consider a literal $p^a(\bar{t})$ in which the adornment a contains some 'd's. We can *project out* the existential arguments by replacing $p^d(\bar{t})$ by $p^d(t\bar{1})$ where the vector of arguments $t\bar{1}$ is obtained from the vector $\bar{t}$ by dropping the arguments corresponding to 'd's. (Note that the length of adornment a is now greater than the number of arguments in $t\bar{1}$. To establish the proper correspondence between the elements of the adornment string a and the vector of arguments $t\bar{1}$, we must ignore the 'd's in a.)

We have a simple lemma characterizing such projections.

Lemma 3.2: Consider the adorned program P^{e_ad}. Let every occurrence of an adorned literal $p^a(\bar{t})$, in rule heads and rule bodies, be consistently replaced by $p^a(t\bar{1})$, where $t\bar{1}$ is $\bar{t}$ with the existential ('d') arguments projected out. The new program computes the same set of answers for the query. []

Henceforth, we will assume that this has been done. Such elimination of arguments, especially from recursive predicates, can have significant impact on performance as observed in [Bancilhon and Ramakrishnan 87]. Reducing the arity of recursive predicates was identified there as an important performance factor. Consider the program for transitive closure presented in the beginning of this section. The recursive predicate was unary whereas in the original program it was binary. We note further that the elimination not only reduces the facts produced but also reduces the duplicate elimination cost significantly. However, we show a version of the problem of pushing projections to be undecidable.

Theorem 3.3: Consider a binary chain program and a query of the form p^{dn}. There is an equivalent monadic chain program if and only if the language defined by the context-free grammar corresponding to the binary chain program is regular. The problem of whether there exists an equivalent monadic chain program is therefore undecidable. []

3.3. Discarding rules

The focus of this section is another kind of optimization, which exploits existential arguments by deleting some rules from the program without changing the set of answers computed for the query. However, the problem of deleting an arbitrary rule while preserving equivalence is also undecidable.

Theorem 3.4: [Gaifman 86, Shmueli 87] For a given adorned program, it is undecidable whether the program computes the same set of answers after a literal or a rule has been deleted. []

Although the problem is undecidable, sufficient conditions can be found which enable us to delete literals and rules in many cases. In fact, we can sometimes delete the recursive rules and make a recursive query non-recursive.

Example 3a:

We continue with Example 3. The first rule defining a^{nd} can be deleted, since any tuple generated using this rule can be generated using the next rule. Note that such a deletion would not be possible if the following rule replaced the third rule in the program:

a^{nd}(X) :- p1(X,Z). []

The notion of *uniform equivalence* was introduced by Sagiv [Sagiv 87] in order to develop an algorithm for deleting rules and literals from Datalog programs. It is decidable whether two programs are uniformly equivalent, and uniform equivalence implies equivalence. Thus, a simple

algorithm for deciding whether to delete a rule or a literal is to check whether the program is uniformly equivalent to itself after the deletion [Sagiv 87], as illustrated in Example 4. But using uniform equivalence is often not powerful enough to show that certain rules can be deleted, as we see in Example 5.

Example 4:

We consider the Example 3 again. The first rule of the program may be deleted while preserving uniform equivalence. We establish this as follows. Consider the following ground instance of rule 1:

a^{nd}(x) :- p(x,z), a^{nd}(z).

Thus, x and z are constants. To show that the program without the first rule is uniformly equivalent to the original program, we must show that the program without the first rule can take the body of the ground instance of rule 1 as the input DB and produce the head of the ground instance of rule 1. Thus, the input DB is {*p(x,z)*, a^{nd}(z)}. The second rule can use the fact *p(x,z)* to generate a^{nd}(x). []

Example 5:

Consider the following program:

Original Program

a(X,Y) :- a(X,Z), p(Z,Y).
a(X,Y) :- p(X,Y).

Adorned Program

a^{nd}(X) :- a^{nn}(X,Z), p(Z,Y).
a^{nd}(X) :- p(X,Y).
a^{nn}(X,Y) :- a^{nn}(X,Z), p(Z,Y).
a^{nn}(X,Y) :- p(X,Y).
a^{nd}(X,Y)?

No rule can be deleted from the adorned program without losing uniform equivalence. []

This motivates a different notion of equivalence which allows us to recognize a richer class of optimizations. We return to the problem of discarding rules in later sections.

4. Notions of Equivalence

Intuitively, two programs containing the same query Q are said to be equivalent if they produce the same output for all inputs. By changing our definitions of input and output, we can define different kinds of equivalence.

DB Equivalence:

Input = An instance of the DB in which the IDB predicates must be empty.
Output = The DB defined by the least fixpoint of *P* computed from the input instance.

Query Equivalence:

Input = An instance of the DB in which the IDB predicates must be empty.
Output = The answer for *q* in the least fixpoint of *P* computed from the input instance.

Uniform Equivalence:

Input = An instance of the DB.
Output = The DB defined by the least fixpoint of *P* computed from the input instance.

Uniform Query Equivalence:

Input = An instance of the DB.
Output = The answer for q in the least fixpoint of P computed from the input instance.

DB equivalence and Query equivalence correspond to what is usually referred to as *equivalence*, although the second is more common. Equivalence is undecidable for Datalog under both these definitions. The third definition was introduced by [Sagiv 87], and is decidable for Datalog. *Uniform query equivalence* is explored here. It is similar to uniform equivalence in that no restrictions are placed on the input. It differs from uniform equivalence in that it considers the output to be just the set of tuples corresponding to the query predicate. This seems more reasonable since we don't really care about the extensions computed for the other predicates. In fact, this observation underlies almost all the query optimization techniques in the literature, in particular those which push selections or projections. The following example shows how uniform query equivalence sometimes allows us to delete rules that cannot be deleted using uniform equivalence.

Example 6: Consider the program of Example 5.

Adorned Program

a^{nd}(X) :- a^{nn}(X,Z), p(Z,Y).
a^{nd}(X) :- p(X,Y).
a^{nn}(X,Y) :- a^{nn}(X,Z), p(Z,Y).
a^{nn}(X,Y) :- p(X,Y).
a^{nd}(X)?

After first step

a^{nd}(X) :- a^{nn}(X,Z), p(Z,Y).
a^{nd}(X) :- p(X,Y).
a^{nn}(X,Y) :- p(X,Y).
a^{nd}(X)?

Optimized Program

a^{nd}(X) :- p(X,Y).
a^{nd}(X)?

The idea is that we only require equivalence with respect to the query predicate a^{nd}. In testing whether the third rule can be deleted, we get the ground instance:

a^{nn}(x,y) :- a^{nn}(x,z), p(z,y).

We test to see if the program without this rule, running on the ground instance of the body as input, produces a^{nd}(x) (rather than a^{nn}(x,y)). If so, we can delete this rule. This is indeed the case, and we get the program shown ('after first step'). Proceeding similarly, we can delete the last rule also. This allows us to delete the first rule since there is now no rule defining a^{nn}. This gives us the optimized program shown above. []

The following simple lemma characterizes these notions of equivalence.

Lemma 4.1: Let $P1$, $P2$ and $G1$, $G2$ be two binary chain programs and their corresponding grammars.

1. $P1$ is *DB equivalent* to $P2$, iff $L(G1,S) = L(G2,S)$ for each non-terminal S in $G1$ or $G2$.
2. $P1$ is *query equivalent* to $P2$, iff $L(G1,Q1) = L(G2,Q2)$, where Qi is the non-terminal corresponding to the query predicate in program Pi.

3. $P1$ is *uniformly equivalent* to $P2$ iff $L^{ex}(G1,S) = L^{ex}(G2,S)$ for each non-terminal S in $G1$ or $G2$.
4. $P1$ is *uniformly query equivalent* to $P2$ iff $L^{ex}(G1,Q1) = L^{ex}(G2,Q2)$, where Qi is the non-terminal corresponding to the query predicate in program Pi. []

It is well-known that DB and Query equivalences are undecidable. From the corresponding undecidability results for context-free grammars [Hopcroft and Ullman 79], it follows that uniform query equivalence is undecidable and thus we have the following lemma, from [Beeri et al. 87].

Lemma 4.2: [Beeri et al. 87] Uniform query equivalence is undecidable for binary chain programs. []

Although uniform query equivalence is undecidable, it allows us to delete rules based on sufficient conditions for uniform query equivalence. This is explored in the next section.

5. An Optimization for Existential Queries Using Uniform Query Equivalence

As we showed in the previous section, uniform query equivalence is undecidable. We now present an algorithm for discarding rules, using a sufficient condition for uniform query equivalence. We begin with some preliminary definitions.

We call a rule of the form $p^a(\bar{t})$:- $p\,1^{a1}(t\bar{1})$ a *unit rule*. The optimization we describe seeks to exploit the presence of such rules. Let us define q^{a1} to *cover* adornment q^a if they are both of the same arity and each n in a corresponds to n in $a1$. Thus it is possible that don't-care (d) arguments of a correspond to n's in $a1$. Intuitively, any tuple in q^{a1} is also a tuple in q^a. We can always add a unit rule $q^a(\bar{t})$:- $q^{a1}(t\bar{1})$, if q^{a1} covers q^a and $t1$ is obtained from $\bar{t}$ by adding a distinct variable (in the appropriate position) for each 'd' in a. (Recall that $\bar{t}$ does not contain any arguments corresponding to the 'd's in a.) In particular, if the query is existential, we can add such unit rules defining it. (With the addition of such rules, the algorithm we present below often captures the essence of "pushing projections").

We define *argument projections* † as follows. Let p^a and $p\,1^{a1}$ be adorned literals. An argument projection $(p^a, p\,1^{a1})$ is a graph with the set of nodes being the 'n' arguments of p^a and $p\,1^{a1}$ For each rule, we obtain an argument projection between the head literal and each derived literal occurrence in the body. We draw an (undirected) edge $(p_i^a, p\,1_j^{a1})$ if the same variable occurs in the ith argument place of p^a and the jth argument place of $p\,1^{a1}$, noting once again that arguments corresponding to a 'd' adornment have been deleted.

Argument projections $(p^a, p\,1^{a1})$ and $(p\,1^{a1}, p\,2^{a2})$ can be composed to yield a composite projection $(p^a, p\,1^{a1})$, $(p\,1^{a1}, p\,2^{a2})$ by merging the corresponding nodes in $p\,1^{a1}$. The summary of a composite projection $(p^a, p\,1^{a1})$, $(p\,1^{a1}, p\,2^{a2})$, ... , $(pn{-}1^{an-1}, pn^{an})$ is an argument projection (p^a, pn^{an}) in which there is an edge between two nodes in the summary if and only if there is a path between the corresponding nodes in the composite mapping.

Without loss of generality, we assume that each literal occurrence in the body of a rule is given a unique occurrence number. Thus, a literal $p(t)$ is renamed $p.n(t)$, where n is the unique occurrence number. It is now easy to identify the rule used to generate an argument projection by looking at the labels of the nodes. (This numbering does not affect the way argument projections are composed - we ignore the numbers while merging nodes.)

Lemma 5.1: Let the query be q^a, and let there be a unit rule $q^a(t)$:- $p.k^c(tk)$. Consider a literal $p.n^c(tn)$. If the summary of every composite argument projection $(q^a, ...)$, ... , $(... , p.n^c)$ is identical to the argument projection $(q^a, p.k^c)$, then we can delete the rule containing $p.n^c$.

---

† This notion is similar to *argument mappings* introduced in [Afrati et al. 86]. However, it is simpler since it only considers equality of arguments, and ignores greater than and lesser than relationships.

Proof (Sketch): Consider a derivation tree (say, D) for a fact $q^a(e)$ which has a fact $p.n^c(e\,1)$ as a node. The subtree (say D1) rooted at $p.n^c(e\,1)$ is a derivation tree for the fact $p^c(e\,1)$. Therefore, there also exists a derivation tree D1' rooted at $p.k^c(e\,1)$, using which we can construct a tree D2 consisting of root $q^a(e)$ and the subtree D1' as the only child. []

If there are no recursive predicates, then it is easy to check if the above theorem allows us to delete a given rule. For each literal occurrence $p.n^c$ in the body such that there is a unit rule $q^a(t)$:- $p.k^c(tn)$, we can generate all appropriate composite argument projections and check if they satisfy the conditions in the corollary. If there are recursive predicates, there may be an infinite number of such composite argument projections to test. However, we only need to generate all possible summaries, and the number of summaries is always finite.

The following simple algorithm generates all possible summaries.

Algorithm 5.1:

Given: A set of argument projections S.

Output: The set of all summaries of composite argument mappings generated from S.

Apply the following until no new summaries can be generated:

1. Every argument projection in S is a summary.
2. The summary of a composition of summaries is also a summary. []

Thus, we have the following simple algorithm for deleting rules.

Algorithm 5.2:

1. Generate all the summaries.
2. Using Lemma 5.1, choose the rules to be discarded. []

As uniform query equivalence is undecidable, this algorithm is not complete in the sense that not all rules that can be deleted will be detected by it. This incompleteness and other aspects of the above optimization are illustrated by the following examples.

Example 7: Consider the following program:

Adorned Program

p^{nd}(X) :- p^{nn}(X,Y).
p^{nd}(X) :- $p\,1^{ndn}$(X,Z), $p\,2^{nnd}$(Z,Y).
p^{nd}(X) :- $b\,1$(X,Y).
p^{nn}(X,Y) :- $p\,1^{ndn}$(X,Z), $p\,2^{nnd}$(Z,Y).
p^{nn}(X,Y) :- $b\,1$(X,Y).
$p\,1^{ndn}$(X,Z) :- p^{nn}(X,U), $b\,2$(U,W,Z).
$p\,1^{ndn}$(X,Z) :- p^{nd}(X), $b\,3$(U,W,Z).
$p\,2^{nnd}$(Z,U) :- $b\,4$(Z,U,V).

Renamed Program

p^{nd}(X) :- $p.\,1^{nn}$(X,Y).
p^{nd}(X) :- $p\,1.2^{ndn}$(X,Z), $p\,2.3^{nnd}$(Z,Y).
p^{nd}(X) :- $b\,1$(X,Y).
p^{nn}(X,Y) :- $p\,1.4^{ndn}$(X,Z), $p\,2.5^{nnd}$(Z,Y).
p^{nn}(X,Y) :- $b\,1$(X,Y).
$p\,1^{ndn}$(X,Z) :- $p.\,6^{nn}$(X,U), $b\,2$(U,W,Z).
$p\,1^{ndn}$(X,Z) :- $p.\,7^{nd}$(X), $b\,3$(U,W,Z).
$p\,2^{nnd}$(Z,U) :- $b\,4$(Z,U,V).

The query is p^{nd}(X,Y)_? and $b1$, $b2$, $b3$ and $b4$ are base relations. By renaming occurrences uniquely, we obtain the renamed program shown above. In subsequent examples, we will present the renamed program directly.

By considering composite argument projections of the form (p^{nd}, ...), ... , (... , $p.6^{nn}$), we find that the summary is identical to the projection (p^{nd}, $p.1^{nn}$) since we ignore the edge between the second arguments. This allows us to discard the sixth rule, and similarly we can discard the seventh rule. (The unit rule used in this case is the trivial rule p^{nd}(X) :- p^{nd}(X).) Having done this, we can discard the second and fourth rule since there are now no rules defining $p\,1^{ndn}$. We can then discard the last rule also. Thus the above program reduces to:

p^{nd}(X) :- $p.1^{nn}$(X,Y).
p^{nd}(X) :- $b\,1$(X,Y).
p^{nn}(X,Y) :- $b\,1$(X,Y).

Note that even though the second rule can be discarded, the above procedure for deleting rules is incapable of doing this. []

Example 8:

The only recursive predicate in the previous examples was the query predicate. This example illustrates the optimization in the presence of other recursive predicates.

p^{nd}(X) :- $p.1^{nn}$(X,Y).
p^{nd}(X) :- $p\,1.2^{nnn}$(X,Z,U), $p\,2.3^{nnd}$(Z,U).
p^{nn}(X,Y) :- $p\,1.4^{nnn}$(X,Z,U), $p\,2.5^{nnn}$(Z,U,Y).
$p\,1^{nnn}$(X,Z,U) :- $p\,1.6^{nnn}$(X,W,W), $g\,1$(W,Z,U).
$p\,1^{nnn}$(X,Z,U) :- $p.7^{nn}$(X,V), $g\,2$(V,Z,U).
$p\,2^{nnd}$(X,Z,U) :- $g\,3$(X,Z,U).
$p\,2^{nnn}$(X,Z,U) :- $g\,3$(X,Z,U).

$g\,1$, $g\,2$ and $g\,3$ are base predicates. The fifth rule can be dropped by considering the literal $p.7$ and the first rule, and using Lemma 5.1. The fourth rule can now be dropped since there is no exit rule defining $p\,1^{nnn}$. This allows us to drop rules 2 and 3 since they contain an undefined predicate ($p\,1^{nnn}$), and similarly, we can then drop rule 1 - i.e., the set of answers is seen to be empty. (We also detect that the last two rules can be dropped after dropping rules 2 and 3, since $p\,2^{nnd}$ and $p\,2^{nnn}$ are then not reachable from the query.) []

Example 9:

This is an example where rules can indeed be deleted using uniform query equivalence, but our technique does not recognize this.

p^{nd}(X) :- p^{nn}(X,Y), $g\,3$(Y,Z,U).
p^{nd}(X) :- $p\,1^{nnn}$(X,Z,U), $g\,1$(Z,U,Y).
$p\,1^{nnn}$(X,Z,U) :- $p\,1^{nnn}$(X,W,W), $g\,2$(W,Z,U).
$p\,1^{nnn}$(X,Z,U) :- p^{nn}(X,V), $g\,3$(V,Z,U), $g\,4$(U,W).

The fourth rule can be deleted although there is no unit rule as before. (We could have added a unit rule, but we chose not to do so in order to illustrate the point.) The reason is that the additional literals in the deleted rule cover the additional literals in the "unit" rule. We discuss how this example can be handled in Example 11. []

We now consider how we can develop weaker sufficient conditions for deleting rules. We first present a generalization of the ideas underlying Lemma 5.1, in a framework which makes explicit certain simplifications that we make in our search for weaker sufficient conditions. We then consider special cases within this framework which allow us to devise inexpensive sufficient

conditions based on summaries.

Given a program $P = (Q, EDB, IDB)$, let us define *an optimistic derivation* as follows. The set of *known facts* is initially the EDB. Consider a rule

$$r: p_0(\bar{X}_0) :- p_1(\bar{X}_1), \ldots, p_n(\bar{X}_n)$$

If there is some ground instance of this rule such that at least one of the $p_i(\bar{X}_i)$, i > 0, is instantiated to a known fact, then, $p_0(\bar{X}_0)$ is added to the set of known facts. We refer to this as a *derivation* of $p_0(\bar{X}_0)$ from $p_i(\bar{X}_i)$ *assuming* $p_j(\bar{X}_j)$, j > 0, j <> i. The *optimistic answer* to query $Q = q(\bar{c},\bar{X})$ is the set of facts of the form $q(\bar{c},\bar{d})$ which can be derived from the EDB using optimistic derivations. (The constant vector of terms $\bar{d}$ must unify with $\bar{X}$).

Theorem 5.2: Consider a set of rules IDB_1, and a query Q. Let r be a rule in IDB_1, and let EDB_1 be the set of facts in the body of a (fixed) ground instance of r. Let IDB_2 be some subset of $(IDB_1 - \{r\})$. If the optimistic answer for (Q, EDB_1, IDB_1) is a subset of the (non-optimistic) answer for (Q, EDB_1, IDB_2), then IDB_1 is uniformly query equivalent to IDB_2. []

Lemma 5.1 is a special case of Theorem 5.2, which uses summaries to test the conditions of Theorem 5.2 efficiently. Lemma 5.1 may fail to show some programs equivalent after deleting a rule while Theorem 5.2 succeeds. However, testing Theorem 5.2 directly may be very expensive (although it always terminates if the domain is finite), and so we will concentrate on efficient tests based on summaries which strengthen Lemma 5.1. We remark that Theorem 5.2 is itself only a sufficient condition, and can be strengthened by considering (modified) optimistic derivations for the program without the rule to be deleted. In this paper, we will confine ourselves to summary based tests which are all subsumed by Theorem 5.2.

Lemma 5.1 essentially corresponds to Theorem 5.2 when the subset of rules IDB_2 is a single unit rule. We now generalize this to the case when the subset contains several unit rules.

Lemma 5.3: Let the query be q^a, and let there be a set of unit rules S in the IDB. Clearly, there is an argument projection corresponding to each rule in S. Let us call this set of argument projections S1. Further, let S2 be the set of summaries produced by applying Algorithm 5.1 to S1. Consider a literal $p.n^c(tn)$. If the summary of every composite argument projection $(q^a, ...), ..., (..., p.n^c)$ is identical to some summary in S2, then we can delete the rule containing $p.n^c$. []

Algorithm 2 can easily be altered to use Lemma 5.3 instead of Lemma 5.1.

Example 10:

Consider:

$p^{nd}(X,Y) :- p^{nn}(X,Y).$
$p^{nd}(X,Y) :- p^{nn}(Y,X).$
$p^{nn}(X,Y) :- q^{nn}(X,Y).$
$p^{nn}(X,Y) :- q^{nn}(Y,X).$
$q^{nn}(X,Y) :- p^{nn}(X,Y).$

The last rule (and therefore, rule 3 and rule 4 as well) can be deleted using Lemma 5.3, but not using Lemma 5.1. []

Summaries essentially limit us to dealing with unit rules. Some slight generalizations are possible, however. We illustrate this in the following example.

Example 11:

Consider the program in Example 9:

$p^{nd}(X)$:- $p^{nn}(X,Y)$, $g\,3(Y,Z,U)$.
$p^{nd}(X)$:- $p\,1^{nnn}(X,Z,U)$, $g\,1(Z,U,Y)$.
$p\,1^{nnn}(X,Z,U)$:- $p\,1^{nnn}(X,W,W)$, $g\,2(W,Z,U)$.
$p\,1^{nnn}(X,Z,U)$:- $p^{nn}(X,V)$, $g\,3(V,Z,U)$, $g\,4(U,W)$.

There is no unit rule. (Recall that in this example, we avoided introducing the unit rule $p^{nd}(X)$:- $p^{nn}(X,Y)$ in order to illustrate the point. In general, we would always add such unit rules. The ideas discussed here apply when the given unit rules do not allow us to delete some rule, and we wish to consider non-unit rules also to see if the deletion is possible.)

Let us rewrite the above program:

$p^{nd}(X)$:- $q\,1^{nnnn}(X,Y,Z,U)$.
$q\,1^{nnnn}(X,Y,Z,U)$:- $p^{nn}(X,Y)$, $g\,3(Y,Z,U)$.
$p^{nd}(X)$:- $p\,1^{nnn}(X,Z,U)$, $g\,1(Z,U,Y)$.
$p\,1^{nnn}(X,Z,U)$:- $p\,1^{nnn}(X,W,W)$, $g\,2(W,Z,U)$.
$p\,1^{nnn}(X,Z,U)$:- $q\,1^{nnnn}(X,V,Z,U)$, $g\,4(U,W)$.

Lemma 5.3 now allows us to delete the last rule. (In fact, Lemma 5.1 allows us to do so as well.) The crucial step of rewriting as in this example is essentially a guess. []

6. Directions for Future Research

We have presented a framework for pushing projections into recursive queries, and introduced the notion of uniform query equivalence. Since both the argument reduction and rule deletion problems are undecidable, and remain so under uniform query equivalence, only sufficient conditions for pushing projections were given. Therefore, we expect that developing more general sufficient conditions will be of interest in this area of optimization. In this context, we discuss some ideas that seem promising.

We make the important observation that projecting out arguments could make some rules, including recursive rules, redundant. While it may be argued that most programs, as originally written, do not permit equivalence preserving deletion of rules, queries frequently project out arguments. Consequently, our observation provides a strong argument for developing algorithms to delete rules under equivalence. We examined the use of Sagiv's algorithm, based on uniform equivalence, and showed its limitations. We introduced the notion of uniform query equivalence and developed sufficient conditions for deleting rules by using it. The rule deletion algorithm complements Sagiv's algorithm.

In Example 9 we observed a weakness of this summary-based algorithm but noted that the fourth rule can indeed be deleted under uniform query equivalence. This is because the first rule subsumes the tuples generated by the fourth rule. So the problem is to devise techniques to detect subsumption of a rule by other rules. Whereas we have restricted our attention to the case of subsumption by a set of (unit) rules, the generalization to the case where a rule is subsumed by a set of (arbitrary) rules is an interesting open question. In particular, can we develop sufficient conditions for uniform query equivalence which are also necessary for uniform equivalence?

Another research direction is to generalize the above results to the more general case of Horn clause queries that include function symbols, evaluable functions, negation, set operations etc. Once again, the problem is to devise sufficient conditions that are useful in pushing projection in the presence of these new operators.

Note that pushing of selection was achieved in optimization algorithms such as Magic Sets, and Counting by transforming the rules that otherwise would not allow 'pushing of selection'. In this paper we did not explore the possibility of rule transformations except for possibly deleting

argument positions. An interesting problem is to explore more general transformations that possibly add literals to (or delete literals from) the rule bodies. We present an illustrative example below.

Example 12:

Consider the following program in which *up*, *dn*, *e* and *b* are all base relations:

Adorned Program

query(X,Y) :- p^{nnd}(X,Y,Z).
p^{nnd}(X,Y,Z) :- p^{nnn}(X,Y,Z).
p^{nnn}(X,Y,Z) :- up(X,X1), p^{nnn}(X1,Y1,Z), dn(Y1,Y), e(Z).
p^{nnn}(X,Y,Z) :- b(X,Y,Z).

Transformed Program

query(X,Y) :- p^{nnd}(X,Y,Z).
query(X,Y) :- b(X,Y,Z).
p^{nnd}(X,Y) :- up(X,X1), p^{nnd}(X1,Y1), dn(Y1,Y).
p^{nnd}(X,Y) :- b(X,Y,Z), e(Z).

As the adorned program requires the literal p^{nnn}, the process of pushing projection is not very useful. However, note that the transformed program has reduced the arity of the recursive predicate by a transformation that preserves uniform query equivalence. []

7. References

[Afrati et al. 86] "Convergence of Sideways Query Evaluation", F. Afrati, C. Papadimitriou, G. Papageorgiou, A. Roussou, Y. Sagiv and J.D. Ullman, *Proc. 5th PODS, 1986.*

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

[Bancilhon et al. 86] "Magic Sets and Other Strange Ways to Implement Logic Programs", F. Bancilhon, D. Maier, Y. Sagiv and J. Ullman, *Proc. 5th PODS, 1986.*

[Bancilhon and Ramakrishnan 86] "An Amateur's Introduction to Recursive Query Processing Strategies", F. Bancilhon and R. Ramakrishnan, *Proc. SIGMOD 86.*

[Bancilhon and Ramakrishnan 87] "Performance Evaluation of Data Intensive Logic Programs", F. Bancilhon and R. Ramakrishnan, *To appear in Foundations of Deductive Databases and Logic Programming, Ed. J. Minker, Morgan Kaufman.*

[Beeri et al. 87] "Sets and Negation in a Logic Database Language (LDL1)," C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmueli and S. Tsur, *Proc. 6th PODS, 1987.*

[Beeri and Ramakrishnan 87] "On the Power of Magic," C. Beeri and R. Ramakrishnan, *Proc. 6th PODS, 1987.*

[Gaifman 86] *Personal communication, reported by Y. Sagiv.*

[Hopcroft and Ullman 79] "Introduction to Automata Theory, Languages, and Computation," J. E. Hopcroft and J. D. Ullman, *Addison-Wesley, 1979.*

[Kifer and Lozinskii 85] "Query Optimization in Logical Databases," M. Kifer and E. L. Lozinskii, *Technical Report 85/16, Dept. of Computer Science, SUNY at Stony Brook.*

[Lloyd 84] "Foundations of Logic Programming," J. W. Lloyd, *Springer-Verlag, 1984.*

[Mendelzon 85] "Functional Dependencies in Logic Programs," A. Mendelzon, *Proc. VLDB 85.*

[Ramakrishnan et al. 87] "Safety of Horn Clauses with Infinite Relations", R. Ramakrishnan, F. Bancilhon and A. Silberschatz, *Proc. 6th PODS, 1987.*

[**Shmueli 87**] ''Decidability and Expressiveness Aspects of Logic Queries'', O. Shmueli, *Proc. 6th PODS, 1987.*

[**Ullman 85**] ''Implementation of Logical Query Languages for Databases,'' J. D. Ullman, *TODS, Vol. 10, No. 3, 1985.*

[**Zaniolo 86**] ''Safety and Compilation of Non-Recursive Horn Clauses'', C. Zaniolo, *Proc. Intl. Conf. on Expert Database Systems, Charleston, 1986.*