Efficiently Ordering Subgoals with Access Constraints

[Extended Abstract]

Guizhen Yang
Artificial Intelligence Center
SRI International
Menlo Park, CA 94025, USA

Michael Kifer
Dept. of Computer Science
Stony Brook University
Stony Brook, NY 11794, USA

Vinay K. Chaudhri
Artificial Intelligence Center
SRI International
Menlo Park, CA 94025, USA

ABSTRACT

In this paper, we study the problem of ordering subgoals under binding pattern restrictions for queries posed as nonrecursive Datalog programs. We prove that despite their limited expressive power, the problem is computationally hard — PSPACE-complete in the size of the nonrecursive Datalog program even for fairly restricted cases. As a practical solution to this problem, we develop an asymptotically optimal algorithm that runs in time linear in the size of the query plan. We also study extensions of our algorithm that efficiently solve other query planning problems under binding pattern restrictions. These problems include conjunctive queries with nested grouping constraints, distributed conjunctive queries, and first-order queries.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Processing*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms

Algorithms, Theory

Keywords

query, nonrecursive Datalog, binding pattern, executability

1. INTRODUCTION

Ordering subgoals under binding pattern restrictions is an important problem of practical significance in information integration and query answering systems [9, 6, 8, 17, 7, 10]. Often binding pattern restrictions are used to define the access constraints associated with information sources. More specifically, these restrictions specify which arguments of a subgoal must be bound to concrete values in order for it to be evaluable. Therefore, query answering under binding pattern restrictions amounts to finding an *executable query plan* — a particular order of the subgoals needed to answer a given query — such that the access constraints associated with each subgoal will be satisfied if evaluation of these subgoals is carried out in the specified order.

The need for binding pattern restrictions arises in a number of situations. Query interfaces on the Web that are based on HTML forms typically require that certain fields be filled in before the query can be dispatched to the backend data source [6, 8, 18, 17]. Thus queries involving joins over multiple such data sources must be evaluated in a manner that respects their binding pattern restrictions. In traditional query optimization, some attributes in a relation may have efficient indexes or be more selective. These sets of attributes can be viewed as binding pattern restrictions, and arranging joins so that these restrictions are satisfied is a good query planning heuristic. In deductive databases and knowledge base systems, some predicates naturally have binding pattern restrictions because their underlying relations are infinite (e.g., the $\geq$ relation) or because these predicates are builtins, which are implemented via foreign procedures that require certain arguments to be bound (e.g., a builtin that checks if a file is writable would normally require the name of the file be given).

In this paper, we study the problem of finding executable query plans for queries posed as *nonrecursive* Datalog programs under binding pattern restrictions. We prove that the associated decision problem is PSPACE-complete in the size of the nonrecursive Datalog program even for fairly restricted cases. Moreover, it is #P-complete to count all the (minimal) binding patterns permissible for conjunctive queries under binding pattern restrictions, thereby providing a complexity-theoretic explanation for the exponential-time algorithm proposed in [18].

Then we proceed to develop what can be viewed as an asymptotically optimal query planning algorithm — an algorithm that runs in time *linear* in the size of the query plan and thus possesses the desirable properties of *output polynomial* algorithms [12]. To clarify a seeming contradiction between this claim and our earlier statement regarding the complexity of the problem, we note that the size of a query plan may be exponential relative to the underlying Datalog program (the proof of Theorem 4 contains such an example). Therefore, although our algorithm runs in time linear in the size of the query plan, its time complexity may still be exponential in the size of the Datalog program.

We also investigate several extensions of our algorithm that efficiently solve other query planning problems under binding pattern restrictions. The first extension is for conjunctive queries with *grouping constraints*, which require that certain groups of subgoals be dispatched together to

PODS'06, June 26–28, 2006, Chicago, Illinois, USA.

one source for evaluation. Query evaluation using views is one instance in which grouping constraints are useful [13, 6]. Such constraints also often arise in integration of heterogeneous knowledge and reasoning systems [2, 1]. We show how to adapt our algorithm to handle grouping constraints, albeit the algorithm now runs in time quadratic in the size of the query. The second extension is for conjunctive queries over horizontally partitioned distributed databases [14]. The main problem here is to decide if a query is *partially* or *fully* executable, especially when different sources may impose different binding pattern restrictions for the same predicate. We show that both problems can be solved efficiently using our algorithm. However, when a query is not fully but only partially executable, it becomes #P-complete to count the number of executable subquery plans. Finally, we show how to extend our algorithm to handle nonrecursive Datalog programs with safe negation without loss of runtime efficiency. Since first-order queries can be translated into nonrecursive Datalog programs with safe negation [5], we solve the open problem of executability (called orderability in [10]) of first-order queries under binding pattern restrictions.

2. PRELIMINARIES

Here we introduce the basic concepts and notations that will be used throughout the paper.

2.1 Rules and Binding Patterns

Let p be an n-ary predicate. We will say $p(X_1, \ldots, X_n)$ is a *goal* or a *subgoal* of predicate p, where each X_i is a variable or a constant. A Datalog program is a finite set of Horn rules without function symbols. Following the standard convention, we will write a rule as follows:

$$h(\overline{X}) \ :- \ g_1(\overline{X_1}), \ldots, g_k(\overline{X_k})$$

in which h and $g_1, \ldots, g_k$ are predicate symbols, and $\overline{X}$, $\overline{X_1}, \ldots, \overline{X_k}$ represent lists of arguments. We will say that predicate h is *defined* by this rule, and call $h(\overline{X})$ the *head* of the rule, and the *conjunction* of literals (separated by commas), $g_1(\overline{X_1}), \ldots, g_k(\overline{X_k})$, the *body* of the rule. We will refer to each $g_i(\overline{X_i})$ as a *subgoal* in the rule.[1] In this paper we assume that all rules are *safe*, i.e., all variables appearing in the head of the rule also appear in some subgoal in the rule body.

In a Datalog program we distinguish between two kinds of predicates: *intensional* predicates, i.e., those predicates that are defined by rules, and *extensional* predicates — predicates that do not appear in the head of any rule. Without loss of generality, we will assume that the sets of intensional and extensional predicates in a Datalog program are disjoint.

Given a Datalog program, we can construct a dependency graph as follows. There is a node in the graph for each predicate in the program. If predicate p is defined by a rule in which a subgoal of predicate q appears, then draw an edge from q to p. A Datalog program is called *recursive* if its dependency graph contains cycles, and called *nonrecursive* otherwise. Note that in this paper we consider only nonrecursive Datalog programs.

We use binding patterns to specify which arguments of a goal are bound, i.e., have a constant value. Furthermore, we consider binding patterns in the context of a set of variables whose bindings (values) are presumed to be available.

Definition 1 (Binding Patterns) Let S be a set of variables and g a goal of an n-ary predicate. The binding pattern for g in the context of S, denoted $\Gamma_S(g)$, is a string of length n consisting of b's and f's. The i-th symbol of $\Gamma_S(g)$ is b if the i-th argument of g is a constant, or a variable that belongs to S; otherwise, the i-th symbol is f. Without a context, the native binding pattern for g, denoted $\Gamma(g)$, is defined as $\Gamma(g) \stackrel{\text{def}}{=} \Gamma_\emptyset(g)$.

We assign to each extensional predicate a set of *feasible* binding patterns. More than one feasible binding pattern can be associated with any predicate. We will say that a subgoal, g, of an extensional predicate, p, is *executable*, if its native binding pattern, $\Gamma(g)$, is feasible for p. We will also say that g is *executable in the context of* S, if the binding pattern of g in the context of S, $\Gamma_S(g)$, is feasible for p.

Example 1 Suppose the ternary predicate, *salary*, has only one feasible binding pattern, bbf. Then $salary(tom, Y, S)$ is not executable, since $\Gamma(salary(tom, Y, S)) = bff \neq bbf$.[2] This binding pattern restriction implies that we can retrieve the salary information of an employee in a year, provided that this employee and the year are both known. However, $salary(tom, Y, S)$ becomes executable in the context of $\{Y\}$, since $\Gamma_{\{Y\}}(salary(tom, Y, S)) = bbf$. □

Since intensional predicates are defined over extensional predicates using rules, feasibility of binding patterns for intensional predicates can be inferred given the feasibility of binding patterns for extensional predicates.

Example 2 Consider the following rule:

$$p(X, Z) \ :- \ s(X, Y), t(Z, Y)$$

and the feasible binding patterns, bf and fb, for extensional predicates s and t, respectively. The binding pattern bf is feasible for predicate p, since we can use bindings for variable X and evaluate $s(X, Y)$ first, obtain bindings for variable Y, and then evaluate $t(Z, Y)$. However, it can be verified that the binding pattern fb is not feasible, since we cannot find an order to execute the subgoals in the rule body that observes their binding pattern restrictions. □

Since feasibility of binding patterns implies executability of subgoals, we need the notion of executable query plans, which we will introduce next.

2.2 Query Plans and Executability

A query plan describes how rules are expanded for subgoals and the order in which subgoals are executed in the expanded rules. For ease of exposition, here we will make the simplifying assumption that the head of a rule does not have duplicate variables in different argument positions.[3]

Let r be a rule, $p(\overline{Y}) \ :- \ q_1(\overline{Y_1}), \ldots, q_k(\overline{Y_k})$, and $p(\overline{X})$ a goal. We assume that r and $p(\overline{X})$ do not share variables,

[1] Frequently we will use the terms "goal" and "subgoal" interchangeably.

[2] Following the standard convention we will use uppercase letters for variables and lowercase for constants.

[3] Any Datalog program can be easily transformed to meet this assumption without affecting the complexity of our algorithms asymptotically.

i.e., rules have only local variables. Let θ be a *substitution* that renames the variables in $p(\overline{Y})$ such that $p(\overline{Y})\theta = p(\overline{X})$, where $p(\overline{Y})\theta$ denotes the result of applying θ to $p(\overline{Y})$. Then the *expansion* of r with respect to $p(\overline{X})$ is the collection of subgoals $q_1(\overline{Y_1})\theta, \ldots, q_k(\overline{Y_k})\theta$.

In the case of nonrecursive Datalog programs, we can represent a *query plan* for a goal or a set (conjunction) of goals using an ordered tree-like data structure.[4] Each node in the tree contains a list of subgoals. The root node contains the original goal or set of goals (which may have been reordered). If the predicate of a subgoal g in a node N is defined by rules $r_1, \ldots, r_m$, then there is a link from g to each node $N_1, \ldots, N_m$, where N_j, $1 \leq j \leq m$, contains the list of subgoals (which may have been reordered) in the expansion of r_j with respect to g_i. We will call g the *parent subgoal* and N the *parent node* of each node N_j, and N_j a *child node* of g and N. Each subtree rooted at N_j is called a *subplan* of g. Each node in the subtree rooted at N_j is called a *descendant node* of g and N.

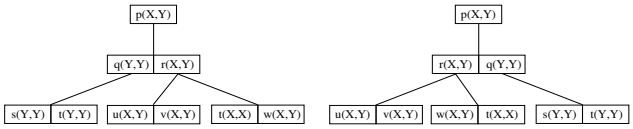

Figure 1: Two Query Plans for Example 3

Note that all the query plans for a goal share the same rule expansion structure, but may differ in the order of subgoals. When we speak of an unordered query plan, we are mainly concerned with subgoals in rule expansions instead of their orderings.

Example 3 Consider the following nonrecursive Datalog program:

$$p(X,Y) :- q(Y,Y), r(X,Y) \qquad r(X,Y) :- u(X,Y), v(X,Y)$$
$$q(X,Y) :- s(X,Y), t(X,Y) \qquad r(X,Y) :- t(X,X), w(X,Y)$$

in which predicates s, t, u, v, w are extensional. Figure 1 shows two example query plans for goal $p(X,Y)$. These two plans have the same rule expansion structure, i.e., are isomorphic when the order of subgoals in each node is not important. But the order of subgoals is different in the rule expansion for $p(X,Y)$, and in the rule expansion for $r(X,Y)$, which is obtained using the second rule above defining predicate r. Note that subgoal $r(X,Y)$ has two subplans. Each subplan corresponds to one rule expansion. □

A query plan is executed top-down (like SLD resolution): (i) the list of subgoals in a node are executed from left to right; (ii) executing an extensional subgoal produces bindings for all the variables in it; (iii) executing an intensional subgoal requires executing all of its subplans (rule expansions) in the same top-down fashion; and (iv) bindings obtained for variables as a result of execution are used to instantiate the same variables in succeeding subgoals.

It is essential that during the top-down execution of a query plan every extensional subgoal be executable given the set of variables that are already bound prior to its execution. In the following, we formally define this set of bound variables based on the evaluation model just described above.

[4]The query plans described here can be viewed as a compact representation of the rule/goal graphs in [15].

Definition 2 (Contexts) Given a query plan T and a set of variables S, let g be a subgoal in a node N of T. The context of g in T with respect to S, denoted $\Phi_{T,S}(g)$, is defined inductively as follows:

$$\Phi_{T,S}(g) \stackrel{\text{def}}{=} \{X \mid X \in varArgs(q), q \in prec(g)\} \cup \Delta$$

where $varArgs(q)$ denotes the set of variable arguments of subgoal q, and $prec(g)$ denotes the set of subgoals that are ordered before g in the same node N. Moreover, $\Delta = S$ if g resides in the root node; otherwise, $\Delta = \Phi_{T,S}(p)$, where p is the parent subgoal of node N.

Note that contexts are defined for *occurrences* of subgoals in a query plan. When we mention a subgoal in a query plan, we are actually referring to the occurrence of the subgoal that resides in a particular node of a query plan. We now formalize the notion of executable query plans as follows.

Definition 3 (Executability) Let T be a query plan and S a set of variables. We will call T an executable query plan in the context of S, if every extensional subgoal g in T is executable in the context of $\Phi_{T,S}(g)$. We will say that a goal g is executable in the context of S, if there is a query plan for g that is executable in the context of S.

We can also extend the notion of executability to a set of subgoals. A set (conjunction) of subgoals $\mathcal{G}$ is *executable in the context of* S, if there is an executable query plan T for the subgoals in $\mathcal{G}$ in the context of S. Note that the root node of T should contain an order of the subgoals in $\mathcal{G}$. We will call this order a *feasible order* of the subgoals in $\mathcal{G}$ in the context of S. Finally, we define the notion of feasible binding patterns for intensional predicates as follows.

Definition 4 (Feasible Binding Patterns) Let p be an n-ary intensional predicate and α a binding pattern of length n. We will say that α is a *feasible* binding pattern for p, if there is an executable query plan for goal $p(X_1, \ldots, X_n)$ in the context of S, where X_i's are distinct variables, and $S = \{X_i \mid \text{the } i\text{-th symbol of } \alpha \text{ is } b\}$.

2.3 The Bound-is-Easier Assumption

We can define a partial order, $\preceq$, among the (feasible) binding patterns[5] of a predicate. Let α and β be two binding patterns of the same predicate. We will write $\alpha \preceq \beta$, if whenever there is a b in α, the corresponding position in β also has a b. For instance, $bbf \preceq bbb$.

Throughout this paper we will make an important assumption, called bound-is-easier [9, 16], about binding patterns: if α is a feasible binding pattern for a predicate and $\alpha \preceq \beta$, then β is also a feasible binding pattern for this same predicate. It can be shown that if the bound-is-easier assumption holds for all extensional predicates, then it also holds for all intensional predicates. Therefore, for any goal g, if g is executable in the context of S_1 and $S_1 \subseteq S_2$, then g is also executable in the context of S_2.

We will say that α is a *minimal* feasible binding pattern for a predicate, if there is no other feasible binding pattern β for this same predicate, such that $\beta \preceq \alpha$. Clearly, to enumerate all the feasible binding patterns of a predicate, it suffices to enumerate all the minimal ones.

[5]Sometimes we will omit the word "feasible" while referring to feasible binding patterns.

3. COMPLEXITY RESULTS

Here we consider the computational complexity of deciding whether a goal defined by a nonrecursive Datalog program is executable in a context (the executability problem). Closely related to this is the problem of deciding whether a binding pattern is feasible for an intensional predicate (the feasibility problem). Since feasibility of binding patterns can be defined in terms of executability of subgoals in particular contexts (see Definition 4), the feasibility problem is essentially equivalent to the executability problem. First we will present a polynomial-space algorithm that is directly targeted at solving the more generic executability problem.

3.1 A Polynomial-Space Algorithm

Algorithm *exec*↓ (p, S) (shown in Figure 2) checks if there is an executable query plan for p in the context of S. When p is an extensional subgoal, the subroutine, *executable*(p, S), on Line 16 is used to test whether p is executable in the context of S. The subroutine, *expand*(p, r), on Line 3 computes the rule expansion, E, of r with respect to p.

```
Algorithm exec↓ (p, S)
input
    p: a goal
    S: a set of variables which are already bound
begin
1.  if p is an intensional subgoal then
2.    for each rule r defining the predicate of p do
3.      E = expand(p, r)
4.      S' = S
5.      while E ≠ ∅ do
6.        if ∃q ∈ E such that exec↓ (q, S') == true then
7.          S' = S' ∪ {all variables in q}
8.          E = E − {q}
9.        else
10.         return false
11.       endif
12.     endwhile
13.   endfor
14.   return true
15. else
16.   return executable(p, S)
end
```

Figure 2: A Polynomial-Space Algorithm for Solving the Executability Problem

Note that algorithm *exec*↓ is essentially constructed based on how executability of query plans is defined in Definition 3. To see why algorithm *exec*↓ works correctly, we need to exploit an important property of the bound-is-easier assumption, which is formally stated in the following theorem.

Theorem 1 ([15]) Let $\mathcal{G} = \{g_1, \ldots, g_n\}$ be a set of subgoals and S a set of variables. Suppose $1 \leq k \leq n$, and each subgoal g_i, $1 \leq i \leq k$, is executable in the context of S_i, where $S_1 = S$, and $S_i \stackrel{\text{def}}{=} S_{i-1} \cup \{\text{all variables in } g_{i-1}\}$ for $2 \leq i \leq k$. Then there is a feasible order of the subgoals in $\mathcal{G}$ in the context of S, iff there is a feasible order of the subgoals in $\mathcal{G}$ in the context of S that starts with $g_1, \ldots, g_k$.

The correctness of algorithm *exec*↓ can be established using Theorem 1 in the following proposition.

Proposition 2 Let p be a goal and S a set of variables. Algorithm *exec*↓ (p, S) returns *true* iff there is an executable query plan for p in the context of S.

To analyze the space complexity of algorithm *exec*↓, let n be the size of the Datalog program. First note that the number of pending calls to algorithm *exec*↓ is $O(n)$. The set, E, of subgoals (Line 3 in Figure 2) in each rule expansion takes space $O(n)$. During each iteration of the while-loop (Lines 5-12 in Figure 2), at most k number of new members may be added to the set, S', of bound variables (Line 7 in Figure 2), where k is the size of the rule currently being expanded in the call. Since the Datalog program is nonrecursive, it follows that inside each call S' takes space $O(n)$. Therefore, algorithm *exec*↓ takes space $O(n^2)$.

We summarize the discussions above in the following claim.

Proposition 3 The executability problem is in PSPACE for nonrecursive Datalog programs.

3.2 Computational Complexity

Here we formally prove that the executability problem is PSPACE-complete for nonrecursive Datalog programs. Note that the claims here use the bound-is-easier assumption. Without this assumption, we can still prove the complexity results here using techniques similar to those in [16], except that each extensional predicate may be allowed to have two different feasible binding patterns.

Theorem 4 For nonrecursive Datalog programs, the executability problem is PSPACE-complete. This complexity result still holds even if every intensional predicate is defined by only one rule, every extensional predicate has only one associated minimal binding pattern, and there are at most two subgoals of the same predicate in a rule.

Corollary 5 The feasibility problem is PSPACE-complete for nonrecursive Datalog programs.

We should point out, however, that the feasibility problem can be solved efficiently for intensional predicates that are defined using extensional predicates only (see Section 4 and Corollary 13). Therefore, an interesting problem one may contend with is that of computing the set of feasible binding patterns for intensional predicates, e.g., using techniques like those proposed in [18]. Such computation requires enumerating all feasible binding patterns. The following theorem states that this enumeration problem is also computationally difficult.

Theorem 6 The problem of counting the number of (minimal) feasible binding patterns for a conjunctive query is #P-complete. This complexity result still holds even if there is no repeated predicate in the query and every predicate in the query has only one associated minimal binding pattern.

Therefore, as far as only one feasibility test is concerned, precomputation of feasible binding patterns offers no computational advantages. We do note, however, that the precomputation approach may be well suited for applications in which a large number of feasibility tests need to be repeated. Although one may argue that the set of feasible binding patterns can be effectively computed by caching the results of single feasibility tests, our focus here is on how to perform single feasibility tests efficiently (see Section 5 for use cases).

$parentGoal(N)$	the parent subgoal of node N
$goalCount(N)$	the number of subgoals that remain to be ordered in node N
$orderedList(N)$	the ordered list of subgoals that are already ordered in node N
$node(g)$	the node to which subgoal g belongs
$ruleCount(g)$	the number of rule expansions that remain to be ordered for subgoal g
$varArgs(g)$	the set of variable arguments of subgoal g
$boundVars(g)$	the set of variables arguments already bound in subgoal g
$isOrdered(g)$	whether subgoal g has been ordered
$goalList(X, N)$	the list of subgoals having variable X as an argument in node N
$isBound(X, N)$	whether variable X is already bound in node N
$isPassed(X, N)$	whether the binding of variable X has been propagated in the subplan rooted at node N
ok	whether an executable query plan has been found
$\mathcal{V}$	a list of variable-node pairs, (X, N), meaning variable X is already bound in node N
$\mathcal{G}$	a list of already ordered subgoals

Figure 3: Summary of Notations and Data Structures

4. IMPROVING TIME EFFICIENCY

Theorem 4 essentially implies it is very unlikely that the executability problem can be solved in time polynomial in the size of the Datalog program. There is a catch, however. Note that executability test basically reduces to verifying if an executable query plan exists. Therefore, a natural question we may ask is: *Can the executability problem be solved efficiently in terms of the size of the query plan?*

The answer is "Yes". But observe that algorithm *exec*↓ tests executability of subgoals in the query plan top-down. In the worst case its running time can be exponential in the size of the query plan. Therefore, the main purpose here is to improve the worst-case time bound of algorithm *exec*↓. We will develop a new algorithm that solves the executability problem in time linear in the size of the query plan.

4.1 A Linear-Time Algorithm

Our new algorithm, named *exec*↑, is shown in Figure 4. The notations used in the algorithm and its subroutines are explained in Figure 3, where we have also briefly described the data structure that each notation represents.

```
Algorithm exec↑ (p, S)
input
    p: a subgoal
    S: a set of variables which are already bound
begin
1.  construct an unordered query plan T for subgoal p
2.  V = G = ∅
3.  R = the root node of T
4.  initialize(R, G)
5.  for each variable X ∈ S do
6.      isBound(X, R) = true
7.      add (X, R) to V
8.  endfor
9.  while V ≠ ∅ or G ≠ ∅ do
10.     while G ≠ ∅ do
11.         select and remove a subgoal q from G
12.         resolve(q, V)
13.     endwhile
14.     while V ≠ ∅ do
15.         select and remove a variable-node pair (X, N) from V
16.         bind(X, N, G)
17.     endwhile
18. endwhile
19. return goalCount(R) == 0
end
```

Figure 4: A Linear-Time Algorithm for Solving the Executability Problem

The key idea underlying algorithm *exec*↑ is to order subgoals in the query plan bottom-up and propagate bindings of variables top-down. It is essentially a greedy algorithm, utilizing the bound-is-easier assumption. Algorithm *exec*↑ consists of three stages: (i) construction of an unordered query plan (refer to Section 2.2 for the definition) for the given goal (Line 1 in Figure 4); (ii) initialization of data structures (Lines 2-8 in Figure 4); and (iii) search for a feasible order of subgoals in the query plan (Lines 9-18 in Figure 4). In the first stage, we simply perform rule expansions and construct an unordered query plan for the given goal. Clearly, the following data structures can be constructed during the construction of the initial query plan. For each node N in the plan: (i) $parentGoal(N)$ is set to the parent subgoal of N; and (ii) $goalCount(N)$ is set to the number of subgoals in N. For each subgoal g in the plan: (i) $node(g)$ is set to the node where g resides; (ii) $ruleCount(g)$ is set to the number of rules expanded for g; and (iii) $varArgs(g)$ is initialized to the set of variable arguments of g.

```
Procedure initialize(N, G)
input
    N: a node in a query plan
    G: a list for storing subgoals
begin
1.  orderedList(N) = ∅
2.  for each subgoal q ∈ N do
3.      boundVars(q) = ∅
4.      isOrdered(q) = false
5.      for each variable X ∈ varArgs(q) do
6.          isBound(X, N) = isPassed(X, N) = false
7.          add q to goalList(X, N)
8.      endfor
9.      if q is an extensional subgoal and executable(q, ∅) then
10.         isOrdered(q) = true
11.         add q to both orderedList(N) and G
12.     endif
13.     for each child node C of q do initialize(C, G) endfor
14. endfor
end
```

Figure 5: Initializing an Unordered Query Plan

In the initialization stage,[6] a main step (Line 4 in Figure 4) is to traverse the initial query plan top-down and build additional data structures needed in the following search stage (see Figure 5). For each node N in the plan, $orderedList(N)$, the list of already ordered subgoals in N, is initialized to empty (Line 1). For each subgoal g in the query plan: (i) $boundVars(g)$, the list of variables already bound in g, is initialized to empty (Line 3); (ii) $isOrdered(g)$, indicating whether g has been ordered, is initialized to *false*

[6]For ease of exposition, we have separated out the initialization stage. In fact, the data structures built in this stage can be constructed together with the initial query plan.

(Line 4); and (iii) g is added to $goalList(X)$, the list of subgoals in which variable X appears as an argument (Line 7). For each variable X appearing in a subgoal in node N: (i) $isBound(X, N)$, indicating if variable X is already bound in N, is initialized to *false* (Line 6); and (ii) $isPassed(X, N)$, indicating whether the binding of variable X has been propagated in the subplan rooted at N, is initialized to *false* (Line 6).

```
Procedure bind(X, N, G)
input
    X: a variable
    N: a node
    G: a list for storing subgoals
begin
1.  if isPassed(X, N) then return endif
2.  isBound(X, N) = isPassed(X, N) = true
3.  for each subgoal q ∈ goalList(X, N) s.t. not isOrdered(q) do
4.    if q is an intensional subgoal then
5.      for each child node C of q do bind(X, C, G) endfor
6.    else
7.      add X to boundVars(q)
8.      if executable(q, boundVars(q)) then
9.        isOrdered(q) = true
10.       add q to both orderedList(N) and G
11.     endif
12.   endif
13. endfor
end
```

Figure 6: Propagating Bindings of Bound Variables

Moreover, we also check, for each extensional subgoal g in node N, if it is executable without any variables to be bound (Line 9). If so, then g is immediately ordered and added to two lists, $orderedList(N)$ and $\mathcal{G}$ (Lines 10-11). Finally, note that in Figure 4, the input, S, to algorithm *exec*↑ is a set of variables that are presumed to be already bound. So for all variable $X \in S$, Lines 5-8 in Figure 4 add a variable-node pair, (X, R), where R is the root node of the query plan, to the list $\mathcal{V}$.

```
Procedure resolve(q, V)
input
    q: a subgoal
    V: a list for storing variable-node pairs
begin
1.  N = node(q)
2.  for each variable X ∈ varArgs(q) s.t. not isBound(X, N) do
3.    isBound(X, N) = true
4.    add (X, N) to V
5.  endfor
6.  goalCount(N) = goalCount(N) − 1
7.  if goalCount(N) == 0 and N is not the root node then
8.    g = parentGoal(N)
9.    ruleCount(g) = ruleCount(g) − 1
10.   if ruleCount(g) == 0 then
11.     isOrdered(g) = true
12.     add g to both orderedList(node(g)) and G
13.   endif
14. endif
end
```

Figure 7: Resolving Ordered Subgoals

The search stage (Lines 9-18 in Figure 4) consists mainly of two interacting subroutines: $bind(X, N, \mathcal{G})$ (Figure 6) and $resolve(q, \mathcal{V})$ (Figure 7). The key idea underlying our search algorithm is based on the following observations. First, once a variable of a subgoal is bound in a node, its binding can be propagated throughout the subplan rooted at that node. Second, once a subgoal becomes executable and is ordered, bindings can be obtained for all of its variable arguments. Third, the bound-is-easier assumption eliminates the need to reorder a subgoal once it has been ordered. Note that the intended use of $\mathcal{G}$ is to store a list of newly ordered subgoals. We use $\mathcal{V}$ to store a list of variable-node pairs. Each $(X, N) \in \mathcal{V}$ means variable X is bound in node N.

The main functionality of procedure $bind(X, N, \mathcal{G})$ (see Figure 6) is to propagate the binding of variable X in the subplan rooted at node N. For each unordered extensional subgoal $q \in goalList(X, N)$, it checks if q becomes executable with the addition of X to the set of variable arguments of q that are already bound (Lines 7-8). If so, then q is immediately ordered and added to $orderedList(N)$ and $\mathcal{G}$ (Lines 10-11). Moreover, the binding of X is propagated further down by calling $bind(X, C, \mathcal{G})$ recursively for each child node C of each unordered, intensional subgoal (Line 5).

Procedure $resolve(q, \mathcal{V})$ (see Figure 7) takes a subgoal q as input, which has been ordered and known to be executable. It first adds to $\mathcal{V}$ a variable-node pair, (X, N), where N is the node in which q resides, for every unbound variable argument X of q (Lines 2-5). The counter, $goalCount(N)$, on Line 6 keeps track of how many subgoals in N remain to be ordered. It is first decremented to account for the ordering of q. When its value is zero, it means a feasible order has been found for all subgoals in N. Lines 7-14 handle the case in which $goalCount(N)$ is zero and N is not the root node. Now N represents a subplan for its parent subgoal g (Line 11). We use the counter, $ruleCount(g)$, on Line 9 to keep track of how many subplans of g remain to be ordered. It is first decremented since a feasible order has been found for N. If its value is zero, it means that all subplans of g are executable. In this case, since g now becomes executable, we order it immediately and add it to $orderedList(node(g))$ and $\mathcal{G}$ (Lines 11-12).

4.2 Correctness and Complexity Analysis

To establish the correctness of algorithm *exec*↑, first observe that a variable-node pair (X, N) is added to $\mathcal{V}$ only if $isBound(X, N)$ equals *false*. Moreover, whenever (X, N) is added to $\mathcal{V}$, $isBound(X, N)$ is set to *true*. Therefore, a variable-node pair is added to $\mathcal{V}$ at most once. Analogously, a subgoal is added to $\mathcal{G}$ at most once. Since every iteration of the outer while-loop on Lines 9-18 in Figure 4 must remove at least one element from either $\mathcal{G}$ or $\mathcal{V}$, it follows that algorithm *exec*↑ must terminate in a finite number of steps.

Theorem 7 Algorithm *exec*↑ (p, S) returns *true* iff an executable query plan exists for p in the context of S.

Corollary 8 An executable query plan for p in the context of S is computed when algorithm *exec*↑ (p, S) returns *true*.

To analyze the running time of algorithm *exec*↑, for now let us assume that the executability test, $executable(g, S)$, where g is an extensional subgoal g and S a set of variables, takes time λ. Let s be the size of the query plan, which is roughly the number of predicate symbols and arguments of all subgoals in the query plan. Clearly, construction of the unordered query plan takes time $O(s)$, and the initialization stage takes time $O(\lambda \cdot s)$. So the cost of executing Lines 1-8 of algorithm *exec*↑ is $O(\lambda \cdot s)$.

The cost of executing the while-loop on Lines 9-18 of algorithm *exec*↑ consists of two parts: the time taken to execute procedure $resolve(q, \mathcal{V})$ (Line 12 in Figure 4) for a subgoal q removed from $\mathcal{G}$, and the time taken to execute procedure $bind(X, N, \mathcal{G})$ (Line 16 in Figure 4) for a variable-node pair (X, N) removed from $\mathcal{V}$. It can be easily verified that each call to procedure $resolve(q, \mathcal{V})$ takes time proportional to the size of q. Since any subgoal can be added to $\mathcal{G}$ at most once, it follows that the accumulative cost of executing Line 12 of algorithm *exec*↑ is $O(s)$.

Note that procedure $bind(X, N, \mathcal{G})$ may be called only in two cases: either due to the removal of (X, N) from $\mathcal{V}$ (Line 12 in Figure 4), or as a result of a recursive call to procedure *bind* itself (Line 5 in Figure 6). However, each variable-node pair (X, N), where variable X appears as a variable argument in a subgoal in node N, is added to $\mathcal{G}$ at most once. The use of $passed(X, N)$ on Lines 1-2 of procedure *bind* makes it impossible to invoke $bind(X, N, \mathcal{G})$ recursively from two different ancestor nodes of N. Therefore, for each variable-node pair (X, N), the number of calls made to $bind(X, N, \mathcal{G})$ can be no more than two. Clearly, Lines 2-13 of procedure $bind(X, N, \mathcal{G})$ may be executed in only one call to $bind(X, N, \mathcal{G})$; the other call simply returns on Line 1 of the procedure. Let us split the cost of executing all calls to procedure *bind* into two parts: ε and ω. For each call to $bind(X, N, \mathcal{G})$: (i) if only Line 1 of the procedure was executed, then we add its cost to ε; (ii) if Lines 2-13 of the procedure were executed, we add its cost to ω, except the cost of executing the for-loop on Line 5 of the procedure — this cost will be tallied separately under the tag of $bind(X, C, \mathcal{G})$. Clearly, $\varepsilon = O(k)$, where k is the number of variable-node pairs, and $\omega = O(\lambda \cdot s)$. But $k = O(s)$. It follows that the accumulative cost of executing Line 16 of algorithm *exec*↑ is $O(\lambda \cdot s)$.

The following theorem summarizes our discussion above about the time complexity of algorithm *exec*↑.

Theorem 9 Algorithm *exec* ↑ takes time $O(\lambda \cdot s)$, where λ is the cost of testing the executability of an extensional subgoal in a given context, and s is the size of the query plan.

The complexity result in Theorem 9 can be interpreted as stating that the number of executability tests is linear in the size of the query plan. Alternatively, we can view λ as representing the cost of executability tests averaged over all extensional subgoals in the query plan. Now let us consider how to manage executability tests efficiently for extensional subgoals.

If the arity of each extensional predicate is small, say, bounded by the length of a machine word, then we can use bitmaps to encode the feasible binding patterns of an extensional predicate as follows: the i-th bit is set to 1 if the i-th argument needs to be bound; otherwise, it is set to 0. On the other hand, the binding pattern of a subgoal is encoded as follows: we set the i-th bit to 0 if the i-th argument is bound, and to 1 otherwise.

Given a subgoal g of an extensional predicate p and a set of variables S, let w be the binding pattern of g in the context of S, and $w_1, \ldots, w_k$ the feasible binding patterns of p that are encoded as bitmaps using the method just described above. Clearly, g is executable in the context of S iff there is w_i, $1 \leq i \leq k$, such that the bit-wise AND of w and w_i equals zero. Therefore, we can precompute the bitmap representations of feasible binding patterns for all extensional predicates, and allocate a machine word for each extensional subgoal to store the bitmap representation of its current binding pattern. Whenever a variable becomes bound (i.e., Line 7 in Figure 6 is executed), we can update the bitmaps accordingly. This step involves retrieving the argument position of a variable in a subgoal (say, using Hash methods) and setting the appropriate bit to 0 (the AND of two machine words). Both operations can be done in constant time. It follows that the executability test for an extensional subgoal can be done in time $O(k)$.

If the arity of each extensional predicate is not bounded by the length of a machine word, however, then we can use a different approach to testing executability of extensional subgoals. Let g be a subgoal of an extensional predicate p, S a set of variables, and $\alpha_1, \ldots, \alpha_k$ the feasible binding patterns for p. We will view each α_i as a set of integers specifying which arguments must be bound. Therefore, we can allocate k counters, $c_1, \ldots, c_k$, to g, each initialized to the number of arguments that remain to be bound as specified by $\alpha_1, \ldots, \alpha_k$, respectively. Whenever a variable $X \in S$ becomes bound (i.e., Line 7 in Figure 6 is executed), we first obtain the argument position, n, of X in g. Then we decrement each c_i if $n \in \alpha_i$. Clearly, g is executable in the context of S iff some c_i ever becomes zero. Assuming set membership tests can done in constant time using Hash methods, we can conclude that the executability test for an extensional subgoal also takes time $O(k)$ even if the arity of each extensional predicate is not bounded.[7]

Summarizing the discussion above, we have the following corollary.

Corollary 10 If the number of feasible binding patterns for each extensional predicate is bounded by a constant, then the executability problem can be solved in time linear in the size of the query plan. The same complexity result holds even if an executable query plan needs to be output when the executability test succeeds.

5. EXTENSIONS

Here we study several different query planning problems that are mainly concerned with ordering subgoals to satisfy binding pattern restrictions. We will show how to extend the algorithms and complexity results developed in Section 4 to these problems. Note that when stating the complexity results in this section, we will implicitly assume that the number of feasible binding patterns for each extensional predicate is bounded. Moreover, for ease of exposition, we will assume that all queries to be considered here contain extensional subgoals only, although our results can be easily extended accordingly to accommodate intensional subgoals.

5.1 Grouping Constraints

A conjunctive query with grouping constraints is like a conventional conjunctive query except that each component in the query may be a group of subgoals. As a syntactic sugar, we will annotate a group of subgoals using a pair of square brackets. For instance, in the query, $q_1 \wedge [q_2 \wedge q_3] \wedge q_4$, the two subgoals, q_2 and q_3, belong to one group. Clearly,

[7]Strictly speaking, we still assume that a machine word is big enough to store the number of arguments in a subgoal.

the case of conjunctive queries with grouping constraints subsumes the case of conventional conjunctive queries. In the latter case, each subgoal can be viewed as belonging to a singleton group (we simply omit the square brackets for singleton groups). Grouping constraints impose restrictions on how subgoals can be ordered — all the subgoals in the same group must remain together. Therefore, given a query, $q_1 \wedge [q_2 \wedge q_3 \wedge q_4] \wedge q_5$, a feasible order (to be formalized next) could be $q_1 \wedge q_5 \wedge [q_2 \wedge q_4 \wedge q_3]$; however, $q_1 \wedge [q_2 \wedge q_4] \wedge q_5 \wedge q_3$ is not valid, since it breaks the grouping constraint.

We allow grouping constraints to have nested structures, i.e., a group of subgoals can be included as a whole in another group. Thus, we define a conjunctive query with grouping constraints (CQG) inductively as follows: (i) if q is a subgoal, then $[q]$ is a CQG; (ii) if $g_1, \ldots, g_n$, $n \geq 2$, are CQGs, so is $[g_1 \wedge \ldots \wedge g_n]$. Note that in our formalization, each subgoal belongs to a singleton group. We normally assume that the groups of subgoals in a CQG are ordered.

We can also define an isomorphism between CQGs inductively based on their nested structures. Let Q_1 and Q_2 be two CQGs. We will say that Q_1 is *isomorphic* to Q_2, if: (i) $Q_1 = [q]$ and $Q_2 = [q]$, where q is a subgoal; or (ii) $Q_1 = [p_1 \wedge \ldots \wedge p_n]$, $Q_2 = [q_1 \wedge \ldots \wedge q_n]$, and there is a permutation, π, of $1, \ldots, n$, such that p_i is isomorphic to $q_{\pi(i)}$, for all $1 \leq i \leq n$. That is, Q_1 has exactly the same grouping of subgoals as Q_2, but the ordering of groups in Q_1 may be different from Q_2.

Given a CQG Q, let *flat*(Q) denote the ordered list of subgoals in Q, i.e., *flat*(Q) is obtained from Q by simply removing its grouping constraints (square brackets). Let S be a set of variables. We will say that Q is *executable* in the context of S, if there is a CQG P such that P is isomorphic to Q, and *flat*(P) is a feasible order of the subgoals in Q in the context of S (recall Definition 3 in Section 2.2).

Procedure $gen(Q, \mathcal{P}_Q)$
input
  Q: a CQG
  $\mathcal{P}_Q$: the Datalog program to be generated from Q
begin
1. **if** Q contains only one subgoal g **then return** g
2. let $Q = [q_1 \wedge \ldots \wedge q_n]$
3. **for** each $q_i, 1 \leq i \leq n$ **do** $g_i = gen(q_i, \mathcal{P}_Q)$
4. generate a new predicate symbol p
5. let $h_Q = p(\overline{X})$, where $\overline{X}$ is the list of variables appearing in Q
6. add to $\mathcal{P}_Q$ the following rule: $h_Q :- g_1, \ldots g_n$
7. **return** h_Q

Figure 8: Generating Datalog Programs from CQGs

The main problem we want to solve here is deciding executability of CQGs. First, given a CQG Q, we will generate a Datalog program, $\mathcal{P}_Q$, from Q using the simple procedure shown in Figure 8. Essentially, procedure $gen(Q, \mathcal{P}_Q)$ produces a rule for each group of subgoals in Q, and uses the heads of these rules as subgoals in rule bodies accordingly. Note that $gen(Q, \mathcal{P}_Q)$ also returns h_Q, the head of the rule finally produced for Q. Let S be a set of variables. It can be easily verified that Q is executable in the context of S, iff h_Q defined by the Datalog program $\mathcal{P}_Q$ is executable in the context of S. Therefore, we can use procedure *gen* and algorithm *exec*$\uparrow$ to solve the executability problem for CQGs.

Note that the Datalog program generated by procedure *gen* is in a special form — what we call *singular* Datalog programs.

Definition 5 (Singular Datalog Programs) We will say that a (nonrecursive) Datalog program $\mathcal{P}$ is *singular*, if for all intensional predicate p in $\mathcal{P}$, there is at most one subgoal of p that appears in the body of at most one rule in $\mathcal{P}$.

Corollary 11 Let g be a subgoal defined by a singular Datalog program, and S a set of variables. Then testing executability of g in the context of S takes time linear in the size of the Datalog program.

Observe that the size of the singular Datalog program $\mathcal{P}_Q$ may be greater than the size of the original query Q. This is due to nested grouping constraints — a variable in an extensional subgoal needs to be copied into the head of the rule for each enclosing group.[8] Let *group*(Q) and *size*(Q) denote the number of nonsingleton groups in Q and the size of Q, respectively. Since procedure *gen* is invoked on every group of subgoals in Q, it follows that the size of $\mathcal{P}_Q$ is $O(group(Q) \cdot size(Q))$. Summarizing the discussion above, we can infer the following claim from Corollary 11.

Corollary 12 Let Q be a conjunctive query with grouping constraints. Then the executability problem for Q can be solved in time $O(group(Q) \cdot size(Q))$.

In fact, it can be shown that $group(Q) = O(size(Q))$. Therefore, in the worst case, the executability problem for CQGs can be solved in time quadratic in the size of the query. Note that if we view a conventional conjunctive query Q as a CQG, then $group(Q) = 1$. Therefore, the following claim immediately follows from Corollary 12.

Corollary 13 The executability problem for conventional conjunctive queries can be solved in time linear in the size of the query.

5.2 Distributed Conjunctive Queries

A distributed conjunctive query (DCQ) is like a conventional conjunctive query except that each subgoal in the query may be distributed among several sources. Given a DCQ, $g_1 \wedge \ldots \wedge g_k$, let us assume that each subgoal g_i is distributed among sources $D_{i1}, \ldots, D_{in_i}$. We will use the notation, $g_i@D_{ij_i}$, to represent the distribution of subgoal g_i at source D_{ij_i}, i.e., $g_i@D_{ij_i}$ is an extensional subgoal to be evaluated at D_{ij_i}. Therefore, the DCQ above can be viewed as semantically equivalent to the following formula in CNF:

$$(g_1@D_{11} \vee \ldots \vee g_1@D_{1n_1}) \wedge \ldots \wedge (g_k@D_{k1} \vee \ldots \vee g_1@D_{kn_k})$$

Clearly, the formula above is equivalent to a union of conjunctive *subqueries* in the form of $g_1@D_{1j_1} \wedge \ldots \wedge g_k@D_{kj_k}$, where $1 \leq j_i \leq n_i$ for all $1 \leq i \leq k$.

Here we will assume that different sources may impose different binding pattern restrictions for the same extensional predicate, but the number of feasible binding patterns is bounded for each extensional predicate at each source. Let $Q = g_1 \wedge \ldots \wedge g_k$ be a DCQ, and S a set of variables. We will say that Q is *fully* executable in the context of S, if for all $1 \leq i \leq k$, $1 \leq j_i \leq n_i$, the subquery, $g_1@D_{1j_1} \wedge \ldots \wedge g_k@D_{kj_k}$, is executable in the context of

[8] This is not the optimal way of generating $\mathcal{P}_Q$ from Q in order to solve the executability problem, but it does not affect our complexity results asymptotically.

S (recall Definition 3 in Section 2.2). We will say that Q is *partially* executable in the context of S, if there exists $1 \leq j_i \leq n_i$ for all $1 \leq i \leq k$, such that the subquery, $g_1@D_{1j_1} \wedge \ldots \wedge g_k@D_{kj_k}$, is executable in the context of S.

Now let us consider how to decide whether a distributed conjunctive query is fully executable. The brute-force approach is to check the executability of every subquery. Since CNF-to-DNF conversion incurs an exponential blowup, it appears that solving this problem would require exponential time.

Not really. Note that rules in Datalog programs represent unions. Given a DCQ, $Q = g_1 \wedge \ldots \wedge g_k$, we can construct a simple nonrecursive Datalog program, $\mathcal{P}_Q$, as follows. First we add to $\mathcal{P}_Q$ the following rule: $g :- g_1, \ldots, g_k$, where g is a new subgoal. Then for each g_i distributed at source D_{ij_i}, we create a rule, $g_i :- g_i@D_{ij_i}$, and add it to $\mathcal{P}_Q$. In the following we will show that Q is fully executable in the context of S, iff subgoal g defined by the Datalog program $\mathcal{P}_Q$ is executable in the context of S, i.e., algorithm $exec\uparrow(g, S)$ returns *true*.

Clearly, when algorithm $exec\uparrow(g, S)$ returns *true*, a feasible order has been computed for subgoals $g_1, \ldots, g_k$. It can be shown that this order is feasible for any subquery of Q in the context of S, by Definition 3. Therefore, Q is fully executable in the context of S. Now suppose $exec\uparrow(g, S)$ returns *false*. Consider the query plan constructed by the algorithm when it terminates. Let $g_{j_1}, \ldots, g_{j_{i-1}}$ be the subgoals that are already ordered, and $g_{j_i}, \ldots, g_{j_k}$ the subgoals that are not yet ordered. Clearly, for each g_{j_m}, $i \leq m \leq k$, there must exist a source $D_{j_m x_m}$ such that $g_{j_m}@D_{j_m x_m}$ is not ordered. It can be easily verified that the subquery $g_{j_1}@D_{j_1 x_1} \wedge \ldots \wedge g_{j_{i-1}}@D_{j_{i-1} x_{i-1}} \wedge g_{j_i}@D_{j_i x_i} \wedge \ldots \wedge g_{j_k}@D_{j_k x_k}$ cannot be executable in the context of S, where $D_{j_m x_m}$ is any source to which g_{j_m} is distributed for $1 \leq m \leq i-1$. Therefore, if $exec\uparrow(g, S)$ returns *false*, then Q is not fully executable in the context of S. Summarizing the discussion here, we have the following claim.

Corollary 14 Checking if a distributed conjunctive query, $g_1 \wedge \ldots \wedge g_k$, is fully executable in a given context takes time $O(\sum_{i=1}^{k} d(g_i) \cdot size(g_i))$, where $d(g_i)$ is the number of sources to which subgoal g_i is distributed, and $size(g_i)$ is the size of g_i.

It is fairly straightforward to check whether a distributed conjunctive query is partially executable in a given context. In fact, we can simply aggregate all the feasible binding patterns for each predicate from all sources, and check if there is a feasible order for the original query with respect to the aggregated binding patterns. Therefore, based on the time complexity analysis in Section 4.2, we can draw the following conclusion.

Corollary 15 Checking if a distributed conjunctive query, $g_1 \wedge \ldots \wedge g_k$, is partially executable in a given context takes time $O(\sum_{i=1}^{k} d(g_i) \cdot size(g_i))$, where $d(g_i)$ is the number of sources to which subgoal g_i is distributed, and $size(g_i)$ is the size of g_i.

We have just shown that the time complexity is about the same for deciding either full or partial executability of distributed conjunctive queries. Both problems can be solved efficiently. What if a distributed conjunctive query is partially executable but not fully executable? In this case, we are faced with the problem of computing all the executable subqueries. This is essentially a *enumeration* problem, which the following theorem indicates is unlikely to be tractable in general.

Theorem 16 It is #P-complete to count the number of executable subqueries of a distributed conjunctive query. This complexity holds even if there are only two sources where the subgoals are distributed, and each predicate has only one minimal feasible binding pattern at each source.

6. RELATED WORK

The problem of ordering subgoals under binding pattern restrictions has been studied extensively [9, 16, 15, 6, 8, 17]. Most closely related to our work here is the work of [9], [16], and [15]. The subgoal ordering algorithm proposed in [9] was targeted at recursive Datalog programs, whereas ours is designed specifically for nonrecursive cases. It was proved in [16] that the executability problem for recursive Datalog programs is EXPTIME-complete. Here we show that the problem is PSPACE-complete when restricted to nonrecursive cases. An earlier study [15] showed that the algorithm proposed in [9] has time complexity $O(n^{2k+5})$, where n is the size of the Datalog program having predicates of maximum arity k. In [15], an algorithm for ordering subgoals in conjunctive queries was presented; it runs in time quadratic in the size of the query. In contrast, our algorithm takes only linear time.

The work of [13, 6, 17, 4] also addressed the problem of querying information sources with access restrictions. But the main challenge there is to compute query plans using views only, which are either semantically equivalent to the original query [13, 17], contained by it [6], or produces the maximal set of answers possible [4]. The polynomial-time algorithm proposed in [4] is capable of generating recursive query plans. However, instead of ordering subgoals explicitly, it uses *domain rules* to overcome binding pattern restrictions. In [18], an exponential-time algorithm was proposed for computing capabilities of information sources. Our complexity results here show that the associated enumeration problem is #P-complete.

More recently, several researchers investigated the problem of deciding executability of queries when query containment needs to be taken into account [7, 11, 10]. In this work, a query is said to be *feasible* if there exists an equivalent, executable query. Therefore, the notion of feasibility can be characterized as *semantic* executability, whereas our algorithm here considers only the *syntactic* form of a query. However, it has been shown that deciding feasibility is as hard as deciding query containment — NP-complete for CQ and UCQ [7], Π_2^P-complete for $CQ^\neg$ and $UCQ^\neg$ [11], and undecidable for recursive Datalog programs [7] and first-order queries in general [10]. Nevertheless, our algorithm can be used by the work of [7, 11, 10] to compute the *answerable* part of a query more efficiently.

In [10], it was proved that deciding orderability (the same as executability here) of first-order queries is NP-complete, if each intensional predicate can be annotated with only one binding pattern. In contrast, our results imply that if multiple annotations are allowed (a much more practical assumption), then the problem can be solved in polynomial time (see Section 7 for more details). Finally, based on an extension of the relational chase theory, [3] proposed a unified

framework for rewriting queries in the presence of views, integrity constraints, and access restrictions. However, the algorithms in [3] are not guaranteed to terminate.

7. DISCUSSION AND CONCLUSION

Note that our algorithm can be extended to handle nonrecursive Datalog programs with safe negation, which requires that all the variables in a rule appear in some positive subgoal in the rule body. That is, negative subgoals are not supposed to "generate" bindings for variables. Thus, we can easily extend our notion of executability here as follows (which is the same as the notion of orderability defined in [10]). Let $\mathcal{G}$ be a set of subgoals and S a set of variables. We use $\mathcal{G}^+$ and $\mathcal{G}^-$ to denote the set of positive and negative subgoals in $\mathcal{G}$, respectively. Then $\mathcal{G}$ is executable in the context of S, if $\mathcal{G}^+$ is executable in the context of S, and all the subgoals in $\mathcal{G}^-$ with negation removed are executable given bindings for all of their variable arguments. Therefore, as far as executability is concerned, negative subgoals can be simply treated like positive subgoals with all variable arguments bound.

It is worth pointing out that our algorithm can still be optimized in several different ways — the way it is presented here is to facilitate our correctness and time complexity analysis. First, it is amenable to parallel implementation, although the degree of parallelism may be limited [16]. In particular, we do not assume any specific execution order on calls to procedure $bind(X, N, \mathcal{G})$ and $resolve(q, \mathcal{V})$ in our correctness analysis. Therefore, these two subroutines may as well be implemented as separate threads that run simultaneously. Second, runtime performance may be further improved if the query plan is only expanded incrementally so as to avoid redundant computation for subgoals having comparable binding patterns. This can be done by maintaining for each predicate a list of binding patterns that have been verified to be feasible. Whenever a new subgoal needs to be expanded, its binding pattern is first checked against these lists. Clearly, there will be no need for expansion if a match is found. Finally, we do note that different orders of executing calls to procedure $bind(X, N, \mathcal{G})$ and $resolve(q, \mathcal{V})$ may exhibit different runtime performance. This scheduling aspect of the algorithm needs further investigation.

There is still one problem that remains open. In [16], it was shown that the executability problem for recursive Datalog programs is EXPTIME-complete. Making things worse, a recursive Datalog query may have multiple executable query plans that are *not* isomorphic to each other in terms of their rule expansion structures (see [16] for an example). Nevertheless, it is not known whether this problem can still be solved in linear time, i.e., an algorithm, upon returning *true*, should only spend time linear in the size of the executable query plan constructed by the algorithm.

Acknowledgments

The work of Michael Kifer was supported in part by NSF grant CCR-0311512, IIS-0534968, and by U.S. Army Medical Research Institute under a subcontract through BNL. The work of Guizhen Yang and Vinay K. Chaudhri was supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCHD030010. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA, or the Department of Interior-National Business Center (DOI-NBC).

8. REFERENCES

[1] J. L. Ambite, V. K. Chaudhri, R. Fikes, J. Jenkins, S. Mishra, M. Muslea, T. Uribe, and G. Yang. Integration of heterogeneous knowledge sources in the CALO query manager. In *ODBASE*, 2005.

[2] Y. Arens, C. A. Knoblock, and W.-M. Shen. Query reformulation for dynamic information integration. *Journal of Intelligent Information Systems (JIIS)*, 6(2/3):99–130, 1996.

[3] A. Deutsch, B. Ludäscher, and A. Nash. Rewriting queries using views with access patterns under integrity constraints. In *ICDT*, 2005.

[4] O. M. Duschka, M. R. Genesereth, and A. Y. Levy. Recursive query plans for data integration. *Journal of Logic Programming (JLP)*, 43(1):49–73, 2000.

[5] A. V. Gelder and R. W. Topor. Safety and translation of relational calculus queries. *ACM Transactions on Database Systems (TODS)*, 16(2):235–278, 1991.

[6] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB*, 1996.

[7] C. Li and E. Y. Chang. On answering queries in the presence of limited access patterns. In *ICDT*, 2001.

[8] C. Li, R. Yerneni, V. Vassalos, H. Garcia-Molina, Y. Papakonstantinou, J. D. Ullman, and M. Valiveti. Capability based mediation in TSIMMIS. In *SIGMOD*, 1998.

[9] K. A. Morris. An algorithm for ordering subgoals in NAIL! In *PODS*, 1988.

[10] A. Nash and B. Ludäscher. Processing first-order queries under limited access patterns. In *PODS*, 2004.

[11] A. Nash and B. Ludäscher. Processing unions of conjunctive queries with negation under limited access patterns. In *EDBT*, 2004.

[12] C. H. Papadimitriou. NP-Completeness: A retrospective. In *International Colloquium on Automata, Languages and Programming (ICALP)*, 1997.

[13] A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns. In *PODS*, 1995.

[14] A. Segev. Optimization of join operations in horizontally partitioned database systems. *ACM Transactions on Database Systems (TODS)*, 11(1):48–80, 1986.

[15] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II: The New Technologies*. Computer Science Press, 1989.

[16] J. D. Ullman and M. Y. Vardi. The complexity of ordering subgoals. In *PODS*, 1988.

[17] V. Vassalos and Y. Papakonstantinou. Expressive capabilities description languages and query rewriting algorithms. *Journal of Logic Programming (JLP)*, 43(1):75–122, 2000.

[18] R. Yerneni, C. Li, H. Garcia-Molina, and J. D. Ullman. Computing capabilities of mediators. In *SIGMOD*, 1999.