On Answering Queries in the Presence of Limited Access Patterns

Chen Li[1] and Edward Chang[2]

[1] Computer Science Department, Stanford University, CA 94305, USA
chenli@cs.stanford.edu
[2] ECE Department, University of California, Santa Barbara, CA 93106, USA
echang@ece.ucsb.edu

Abstract. In information-integration systems, source relations often have limitations on access patterns to their data; i.e., when one must provide values for certain attributes of a relation in order to retrieve its tuples. In this paper we consider the following fundamental problem: can we compute the complete answer to a query by accessing the relations with legal patterns? The *complete* answer to a query is the answer that we could compute if we could retrieve all the tuples from the relations. We give algorithms for solving the problem for various classes of queries, including conjunctive queries, unions of conjunctive queries, and conjunctive queries with arithmetic comparisons. We prove the problem is undecidable for datalog queries. If the complete answer to a query cannot be computed, we often need to compute its maximal answer. The second problem we study is, given two conjunctive queries on relations with limited access patterns, how to test whether the maximal answer to the first query is contained in the maximal answer to the second one? We show this problem is decidable using the results of monadic programs.

1 Introduction

The goal of information-integration systems (e.g., [3, 20, 25]) is to support seamless access to heterogeneous data sources. In these systems, a user poses a query on a mediator [26], which computes the answer by accessing the data at the underlying source relations. One of the challenges for these systems is to deal with the diverse capabilities of sources in answering queries. For instance, a source relation *r(Star,Movie)* might not allow us to retrieve all its data "for free." Instead, the only way of retrieving its tuples is by providing a star name, and then retrieving the movies of this star. In general, relations in these systems may have limitations on access patterns to their data; i.e., one must provide values for certain attributes of a relation to retrieve its tuples. There are many reasons for these limitations, such as restrictive web search forms and concerns of security and performance.

In this paper we first study the following fundamental problem: *Given a query on relations with limited access patterns, can we compute the complete answer to the query by accessing the relations with legal patterns?* The *complete* answer to the query is the answer that we could compute if we could retrieve all the tuples from the relations. Often users make decisions based on whether the answers to certain queries are complete or not. Thus the solution to this problem is important for the decision support and analysis by users. The following example shows that in some cases we can compute the complete answer to a query, even though we cannot retrieve all the tuples from relations.

Example 1. Suppose we have two relations $r(Star, Movie)$ and $s(Movie, Award)$ that store information about movies and their stars, and information about movies and the awards they won, respectively. The access limitation of relation r is that each query to this relation must specify a star name. Similarly, the access limitation of s is that each query to s must specify a movie name. Consider the following query that asks for the awards of the movies in which Fonda starred:

$$Q_1 : ans(A) \text{ :- } r(fonda, M), s(M, A)$$

To answer Q_1, we first access relation r to retrieve the movies in which Fonda starred. For each returned movie, we access relation s to obtain its awards. Finally we return all these awards as the answer to the query. Although we did not retrieve all the tuples in the two relations, we can still claim that the computed answer *is* the complete answer to Q_1. The reason is that all the tuples of relation r that satisfy the first subgoal were retrieved in the first step. In addition, all the tuples of s that satisfy the second subgoal and join with the results of the first step were retrieved in the second step.

However, if the access limitation of the relation r is that each query to r must specify a movie title (not a star name), then we cannot compute the complete answer to query Q_1. The reason is that there can be a movie that Fonda starred, but we cannot retrieve the tuple without knowing the movie.

In general, if the complete answer to a query can be computed for any database of the relations in the query, we say that the query is *stable*. For instance, the query Q_1 in Example 1 is a stable query. As illustrated by the example, we might think that we can test the stability of a query by checking the existence of a *feasible* order of all its subgoals, as in [9, 28]. An order of subgoals is feasible if for each subgoal in the order, the variables bound by the previous subgoals provide enough bound arguments that the relation for the subgoal can be accessed using a legal pattern. However, the following example shows that a query can be stable even if such a feasible order does not exist.

Example 2. We modify query Q_1 slightly by adding a subgoal $r(S, M)$, and have the following query:

$$Q_2 : ans(A) \text{ :- } r(fonda, M), s(M, A), r(S, M)$$

This query does not have a feasible order of all its subgoals, since we cannot bind the variable S in the added subgoal. However, this subgoal is actually redundant, and we can show that Q_2 is equivalent to query Q_1. That is, there is a containment mapping [4] from Q_2 to Q_1, and vice versa. Therefore, for any database of the two relations, we can still compute the complete answer to Q_2 by answering Q_1.

Example 2 suggests that testing stability of a conjunctive query is not just checking the existence of a feasible order of all its subgoals. In this paper we study how to test stability of a variety of queries. The following are the results:

1. We show that a conjunctive query is stable iff its minimal equivalent query has a feasible order of all its subgoals. We propose two algorithms for testing stability of conjunctive queries, and prove this problem is $\mathcal{NP}$-complete (Section 3).
2. We study stability of finite unions of conjunctive queries, and give similar results as conjunctive queries. We propose two algorithms for testing stability of unions of conjunctive queries, and prove that stability of datalog queries is undecidable (Section 3).
3. We propose an algorithm for testing stability of conjunctive queries with arithmetic comparisons (Section 4).
4. We show that the complete answer to a nonstable conjunctive query can be computed for certain databases. We develop a decision tree (Figure 1) to guide the planning process to compute the complete answer to a conjunctive query (Section 5).

In the cases where we cannot compute the complete answer to a query, we often want to compute its maximal answer. The second problem we study is, *given two queries on relations with limited access patterns, how to test whether the maximal answer to the first query is contained in the maximal answer to the second one?* Clearly the solution to this problem can be used to answer queries efficiently.

Given a conjunctive query on relations with limited access patterns, [8, 17] show how to construct a recursive datalog program [24] to compute the maximal answer to the query. That is, we can retrieve tuples from relations by retrieving as many bindings from the relations and the query as possible, then use the obtained tuples to answer the query. For instance, consider the relation $s(Movie, Award)$ in Example 1. Suppose we have another relation $dm(Movie)$ that provides movies made by Disney. We can use these movies to access relation s to retrieve tuples, and use these tuples to answer a query on these relations.

To test whether the maximal answer to a conjunctive query is contained in the maximal answer to another conjunctive query, we need to test whether the datalog program for the first one is contained in that for the second one. Since containment of datalog programs is undecidable [23], our problem of query containment seems undecidable. However, in Section 6 we prove this containment problem is decidable using the results of monadic programs [6]. Our results extend the recent results by Millstein, Levy, and Friedman [19], since we loosen the assumption in that paper. We also discuss how to test the containment efficiently when the program for a query in the test is inherently not recursive.

2 Preliminaries

Limited access patterns of relations can be modeled using *binding patterns* [24]. A binding pattern of a relation specifies the attributes that must be given values ("bound") to access the relation. In each binding pattern, an attribute is adorned as "b" (a value must be specified for this attribute) or "f" (the attribute can be free). For example, a relation $r(A, B, C)$ with the binding patterns $\{bff, ffb\}$ requires that every query to the relation must either supply a value for the first argument, or supply a value for the third argument.

Given a database D of relations with binding patterns and a query Q on these relations, the *complete* answer to Q, denoted by $ANS(Q, D)$, is the query's answer that could be computed if we could retrieve *all* tuples from the relations. However, we may not be able to retrieve all these tuples due to the binding patterns. The following observation serves as a starting point of our work.

> If a relation does not have an all-free binding pattern, then after some finite source queries are sent to the relation, there can always be some tuples in the relation that have not been retrieved, because we did not obtain the necessary bindings.

Definition 1. *(stable query) A query on relations with binding patterns is* stable *if for any database of the relations, we can compute the complete answer to the query by accessing the relations with legal patterns.*

We assume that if a relation requires a value to be given for a particular argument, the domain of the argument is infinite, or we do not know all the possible values for this argument. For example, the relation $r(Star, Movie)$ in Example 1 requires a star name, and we assume that we do not know all the possible star names. As a result, we do not allow the "strategy" of trying all the (infinite) possible strings as the argument to test the relation, since this approach does not terminate. Instead, we assume that each binding we use to access a relation is either from a query, or from the tuples retrieved by another access to a relation, while the value is from the appropriate domain.

Now the fundamental problem we study can be stated formally as follows: *how to test the stability of a query on relations with binding patterns?* As we will see in Section 3, in order to prove a query is stable, we need to show a legal plan that can compute the complete answer to the query for any database of the relations. On the other hand, in order to prove that a query Q is not stable, we need to give two databases D_1 and D_2, such that they have the same observable tuples. That is, by using only the bindings from the query and the relations, for both databases we can retrieve the same tuples from the relations. However, the two databases yield different answers to query Q, i.e., $ANS(Q, D_1) \neq ANS(Q, D_2)$. Therefore, based on the retrievable tuples from the relations, we cannot tell whether the answer computed using these tuples is the complete answer or not.

3 Stability of Conjunctive Queries, Unions of Conjunctive Queries, and Datalog Queries

In this section we develop two algorithms for testing stability of a conjunctive query, and prove this problem is $\mathcal{NP}$-complete. We also propose two algorithms for testing stability of a finite union of conjunctive queries. We prove that stability of datalog queries is undecidable.

3.1 Stability of Conjunctive Queries

A conjunctive query (CQ for short) is denoted by:

$$h(\bar{X}) \text{ :- } g_1(\bar{X}_1), \ldots, g_n(\bar{X}_n)$$

In each subgoal $g_i(\bar{X}_i)$, predicate g_i is a relation, and every argument in $\bar{X}_i$ is either a variable or a constant. The variables $\bar{X}$ in the head are called *distinguished* variables. We use names beginning with lower-case letters for constants and relation names, and names beginning with upper-case letters for variables.

Definition 2. *(feasible order of subgoals) Some subgoals $g_1(\bar{X}_1), \ldots, g_k(\bar{X}_k)$ in a CQ form a* feasible order *if each subgoal $g_i(\bar{X}_i)$ in the order is* executable*; that is, there is a binding pattern p of the relation g_i, such that for each argument A in $g_i(\bar{X}_i)$ that is adorned as b in p, either A is a constant, or A appears in a previous subgoal. A CQ is* feasible *if it has a feasible order of* all *its subgoals.*

The query Q_1 in Example 1 is feasible, since $\big(r(fonda, M), s(M, A)\big)$ is a feasible order of all its subgoals. The query Q_2 is not feasible, since it does not have such a feasible order. A subgoal in a CQ is *answerable* if it is in a feasible order of *some* subgoals in the query. The answerable subgoals of a CQ can be computed by a greedy algorithm, called the Inflationary algorithm. That is, initialize a set Φ_a of subgoals to be empty. With the variables bound by the subgoals in Φ_a, whenever a subgoal becomes executable by accessing its relation, add this subgoal to Φ_a. Repeat this process until no more subgoals can be added to Φ_a, and Φ_a will include all the answerable subgoals of the query. Clearly a query is feasible if and only if all its subgoals are answerable. The following lemma shows that feasibility of a CQ is a sufficient condition for its stability.

Lemma 1. *A feasible CQ is stable. That is, if a CQ has a feasible order of all its subgoals, for any database of the relations, we can compute the complete answer to the query.*[1]

Corollary 1. *A CQ is stable if it has an equivalent query that is feasible.*

[1] We do not provide all the proofs of the lemmas and theorems in this paper due to space limitations. Refer [15, 16] for details.

The query Q_2 in Example 2 is the *minimal equivalent query* of Q_1. A CQ is *minimal* if it has no redundant subgoals, i.e., removing any of its subgoals will yield a nonequivalent query. It is known that each CQ has a unique minimal equivalent up to renaming of variables and reordering of subgoals, which can be obtained by deleting its redundant subgoals [4]. Now we give two theorems that suggest two algorithms for testing the stability of a CQ.

Theorem 1. *A CQ is stable iff its minimal equivalent is feasible.*

By Theorem 1, we give an algorithm CQstable for testing the stability of a CQ Q. The algorithm first computes the minimal equivalent Q_m of Q by deleting the redundant subgoals in Q. Then it uses the Inflationary algorithm to test the feasibility of Q_m. If Q_m is feasible, then Q is stable; otherwise, Q is not stable. The complexity of the algorithm CQstable is exponential, since we need to minimize the CQ first, which is known to be $\mathcal{NP}$-complete [4]. There is a more efficient algorithm that is based on the following theorem.

Theorem 2. *Let Q be a CQ on relations with binding patterns and Q_a be the query with the head of Q and the answerable subgoals of Q. Then Q is stable iff Q and Q_a are equivalent as queries, i.e., $Q = Q_a$.*

Theorem 2 gives another algorithm CQstable* for testing the stability of a CQ Q as follows. The algorithm first computes all the answerable subgoals of Q. If all the subgoals are answerable, then Q is stable, and we do not need to test any query containment. Otherwise, the algorithm constructs the query Q_a with these answerable subgoals and the head of Q. It tests whether Q_a is contained in Q (denoted $Q_a \sqsubseteq Q$) by checking if there is a containment mapping from Q to Q_a. If so, since $Q \sqsubseteq Q_a$ is obvious, we have $Q = Q_a$, and Q is stable; otherwise, Q is not stable. The algorithm CQstable* has two advantages: (1) If all the subgoals of Q are answerable, then we do not need to test whether $Q_a \sqsubseteq Q$, thus its time complexity is polynomial in this case. (2) As we will see in Section 4, this algorithm can be generalized to test stability of conjunctive queries with arithmetic comparisons.

Theorem 3. *The problem of testing stability of a CQ is $\mathcal{NP}$-complete.*

Proof. (sketch) Algorithm CQstable shows that the problem is in $\mathcal{NP}$. It is known that given a CQ Q and a CQ Q' that has a subset of the subgoals in Q, the problem of deciding whether $Q' \sqsubseteq Q$ is $\mathcal{NP}$-complete [4]. It can be shown that this problem can be reduced to our stability problem in polynomial time [15].

3.2 Stability of Unions of Conjunctive Queries

Let $\mathcal{Q} = Q_1 \cup \cdots \cup Q_n$ be a finite union of CQ's (UCQ for short), and all its CQ's have a common head predicate. It is known that there is a unique minimal subset of $\mathcal{Q}$ that is its minimal equivalent [22].

Example 3. Suppose we have three relations r, s, and p, and each relation has only one binding pattern bf. Consider the following three CQ's:

Q_1: $ans(X)$:- $r(a, X)$
Q_2: $ans(X)$:- $r(a, X), p(Y, Z)$
Q_3: $ans(X)$:- $r(a, X), s(X, Y), p(Y, Z)$

Clearly $Q_3 \sqsubseteq Q_2 \sqsubseteq Q_1$. Queries Q_1 and Q_3 are both stable (since they are both feasible), while query Q_2 is not. Consider the following two UCQ's: $\mathcal{Q}_1 = Q_1 \cup Q_2 \cup Q_3$ and $\mathcal{Q}_2 = Q_2 \cup Q_3$. $\mathcal{Q}_1$ has a minimal equivalent Q_1, and $\mathcal{Q}_2$ has a minimal equivalent Q_2. Therefore, query $\mathcal{Q}_1$ is stable, and $\mathcal{Q}_2$ is not.

In analogy with the results for CQ's, we have the following two theorems:

Theorem 4. *Let $\mathcal{Q}$ be a UCQ on relations with binding patterns. $\mathcal{Q}$ is stable iff each query in the minimal equivalent of $\mathcal{Q}$ is stable.*

Theorem 5. *Let $\mathcal{Q}$ be a UCQ on relations with binding patterns. Let $\mathcal{Q}_s$ be the union of all the stable queries in $\mathcal{Q}$. Then $\mathcal{Q}$ is stable iff $\mathcal{Q}$ and $\mathcal{Q}_s$ are equivalent as queries, i.e., $\mathcal{Q} = \mathcal{Q}_s$.*

Theorem 4 gives an algorithm UCQstable for testing stability of a UCQ $\mathcal{Q}$ as follows. Compute the minimal equivalent $\mathcal{Q}_m$ of $\mathcal{Q}$, and test the stability of each CQ in $\mathcal{Q}_m$ using the algorithm CQstable or CQstable*. If all the queries in $\mathcal{Q}_m$ are stable, query $\mathcal{Q}$ is stable; otherwise, $\mathcal{Q}$ is not stable. Theorem 5 gives another algorithm UCQstable* for testing stability of a UCQ $\mathcal{Q}$ as follows. Test the stability of each query in $\mathcal{Q}$ by calling the algorithms CQstable or CQstable*. If all the queries are stable, then $\mathcal{Q}$ is stable. Otherwise, let $\mathcal{Q}_s$ be the union of these stable queries. Test whether $\mathcal{Q} \sqsubseteq \mathcal{Q}_s$, i.e., $\mathcal{Q}$ is contained in $\mathcal{Q}_s$ as queries. If so, $\mathcal{Q}$ is stable; otherwise, $\mathcal{Q}$ is not stable. The advantage of this algorithm is that we do not need to test whether $\mathcal{Q} \sqsubseteq \mathcal{Q}_s$ if all the queries in $\mathcal{Q}$ are stable.

3.3 Stability of Datalog Queries

We want to know that given a datalog query on EDB predicates [24] with binding patterns, can we compute the complete answer to the query by accessing the EDB relations with legal patterns? Not surprisingly, this problem is not decidable.

Theorem 6. *Stability of datalog queries is undecidable.*

Proof. (Sketch) Let P_1 and P_2 be two arbitrary datalog queries. We show that a decision procedure for the stability of datalog queries would allow us to decide whether $P_1 \sqsubseteq P_2$. Since containment of datalog queries is undecidable, we prove the theorem.[2] Let all the EDB relations in the two queries have an all-free binding pattern; i.e., there is no restriction of retrieving tuples from these relations. Without loss of generality, we can assume that the goal predicates in P_1 and P_2, named p_1 and p_2 respectively, have arity m. Let $\mathcal{Q}$ be the datalog query consisting of all the rules in P_1 and P_2, and of the rules:

[2] The idea of the proof is borrowed from [7], Chapter 2.3.

$$r_1\colon ans(X_1,\ldots,X_m) \text{ :- } p_1(X_1,\ldots,X_m), e(Z)$$
$$r_2\colon ans(X_1,\ldots,X_m) \text{ :- } p_2(X_1,\ldots,X_m)$$

where e is a new 1-ary relation with the binding pattern b. Variable Z is a new variable that does not appear in P_1 and P_2. We can show that $P_1 \sqsubseteq P_2$ if and only if query $\mathcal{Q}$ is stable.

In [15] we give a sufficient condition for stability of datalog queries. We show that if a set of rules on EDB relations with binding patterns has a feasible rule/goal graph [24] w.r.t. a query goal, then the query is stable.

4 Stability of Conjunctive Queries with Arithmetic Comparisons

In this section we develop an algorithm for testing the stability of a conjunctive query with arithmetic comparisons (CQAC for short). This problem is more challenging than conjunctive queries, because equalities may help bind more variables, and then make more subgoals answerable. In addition, a CQAC may not have a minimal equivalent formed from a subset of its own subgoals, as shown by Example 14.8 in [24]. Therefore, we cannot generalize the algorithm CQstable to solve this problem.

Assume Q is CQAC. Let $O(Q)$ be the set of ordinary (uninterpreted) subgoals of Q that do not have comparisons. Let $\mathcal{C}(Q)$ be the set of subgoals of Q that are arithmetic comparisons. We consider the following arithmetic comparisons: $<$, $\leq$, $=$, $>$, $\geq$, and $\neq$. In addition, we make the following assumptions about the comparisons: (1) Values for the variables in the comparisons are chosen from an infinite, totally ordered set, such as the rationals or reals. (2) The comparisons are not contradictory, i.e., there exists an instantiation of the variables such that all the comparisons are true. In addition, all the comparisons are safe, i.e., each variable in the comparisons appears in some ordinary subgoal.

Definition 3. *(answerable subquery of a CQAC) The* answerable subquery *of a CQAC Q on relations with binding patterns, denoted by Q_a, is the query including the head of Q, the answerable subgoals Φ_a of Q, and all the comparisons of the bound variables in Φ_a that can be derived from $\mathcal{C}(Q)$.*

The answerable subquery Q_a of a CQAC Q can be computed as follows. We first compute all the answerable ordinary subgoals Φ_a of query Q using the Inflationary algorithm. Note that if Q contains equalities such as $X = Y$, or equalities that can be derived from inequalities (e.g., if we can derive $X \leq Y$ and $X \geq Y$, then $X = Y$), we need to substitute variable X by Y before using the Inflationary algorithm to find all the answerable subgoals. Derive all the inequalities among the variables in Φ_a from $\mathcal{C}(Q)$. Then Q_a includes *all* the constraints of the variables in Φ_a, because $\mathcal{C}(Q)$ may derive more constraints that these variables should satisfy. For instance, assume variable X is bound in Φ_a, and variable Y is not. If Q has comparisons $X < Y$ and $Y < 5$, then variable X in Q_a still needs to satisfy the constraint $X < 5$.

We might be tempted to generalize the algorithm CQstable* as follows. Given a CQAC Q, we compute its answerable subquery Q_a. We test the stability of Q by testing whether $Q_a \sqsubseteq Q$, which can be tested using the algorithm in [11, 29] ("the GZO algorithm" for short). However, the following example shows that this "algorithm" does not always work.

Example 4. Consider query

$$P : ans(Y) \text{ :- } p(X), r(X, Y), r(A, B), A < B, X \leq A, A \leq Y$$

where relation p has a binding pattern f, and relation r has a binding pattern bf. Its answerable subquery is

$$P_a : ans(Y) \text{ :- } p(X), r(X, Y), X \leq Y$$

Using the GZO algorithm we know $P_a \not\sqsubseteq P$. Therefore, we may claim that query P is not stable. However, actually query P *is* stable. As we will see shortly, P is equivalent to the union of the following two queries.

$$T_1 : ans(Y) \text{ :- } p(X), r(X, Y), X < Y$$
$$T_2 : ans(Y) \text{ :- } p(Y), r(Y, Y), r(Y, B), Y < B$$

Note both T_1 and T_2 are stable, since all their ordinary subgoals are answerable.

The above "algorithm" fails because the only case where $P_a \not\sqsubseteq P$ is when $X = Y$. However, $X \leq A$ and $A \leq Y$ will then force $A = X = Y$, and the subgoal $r(A, B)$ becomes answerable! This example suggests that we need to consider *all* the total orders of the query variables, similar to the idea in [12].[3]

Theorem 7. *Let Q be a CQAC, and $\Omega(Q)$ be the set of all the total orders of the variables in Q that satisfy the comparisons of Q. For each $\lambda \in \Omega(Q)$, let Q^λ be the corresponding query that includes the ordinary subgoals of Q and all the inequalities and equalities of this total order λ. Query Q is stable if and only if for all $\lambda \in \Omega(Q)$, $Q_a^\lambda \sqsubseteq Q$, where Q_a^λ is the answerable subquery of Q^λ.*

The theorem suggests an algorithm CQACstable for testing the stability of a CQAC Q as follows:

1. Compute all the total orders $\Omega(Q)$ of the variables in Q that satisfy the comparisons in Q.
2. For each $\lambda \in \Omega(Q)$:
   (a) Compute the answerable subquery Q_a^λ of query Q^λ;
   (b) Test $Q_a^\lambda \sqsubseteq Q$ by calling the GZO algorithm;
   (c) If $Q_a^\lambda \not\sqsubseteq Q$, claim that query Q is not stable and return.
3. Claim that query Q is stable.

[3] Formally, a total order of the variables in the query is an order with some equalities, i.e., all the variables are partitioned to sets $S_1, \ldots, S_k$, such that each S_i is a set of equal variables, and for any two variables $X_i \in S_i$ and $X_j \in S_j$, if $i < j$, then $X_i < X_j$.

For a CQAC Q on a database D, if by calling algorithm CQACstable we know Q is stable, we can compute $ANS(Q, D)$ by computing $ANS(Q_a^\lambda, D)$ for each total order λ in $\Omega(Q)$, and taking the union of these answers.

Example 5. The query P in Example 4 has the following 8 total orders:

λ_1: $X < A = Y < B$; λ_2: $X < A < Y < B$; λ_3: $X < A < Y = B$;
λ_4: $X < A < B < Y$; λ_5: $X = A = Y < B$; λ_6: $X = A < Y < B$;
λ_7: $X = A < Y = B$; λ_8: $X = A < B < Y$.

For each total order λ_i, we write its corresponding query P^{λ_i}. For instance:

$$P^{\lambda_1} : ans(Y) \text{ :- } p(X), r(X, Y), r(Y, B), X < Y, Y < B$$

We then compute the answerable subquery $P_a^{\lambda_i}$. All the 8 answerable subqueries are contained in P. By Theorem 7, query P is stable. Actually, the union of all the answerable subqueries except $P_a^{\lambda_5}$ is:

$$ans(Y) \text{ :- } p(X), r(X, Y), r(A, B), X < Y, A < B, X \leq A, A \leq Y$$

whose answerable subquery is the query T_1 in Example 4. In addition, $P_a^{\lambda_5}$ is equivalent to the query T_2. Since P is equivalent to the union of the 8 answerable subqueries, we have proved $P = T_1 \cup T_2$.

5 Nonstable Conjunctive Queries with Computable Complete Answers

So far we have considered whether the complete answer to a query can be computed for *any* database. In this section we show that for *some* databases, even if a CQ is not stable, we may still be able to compute its complete answer. However, the computability of its complete answer is data dependent, i.e., we do not know the computability until some plan is executed. The following is an example.

Example 6. Consider the following queries on relations with binding patterns:

Relations	**Binding patterns**	**Queries**	
$r(A, B, C)$	bff	$Q_1 : ans(B)$	:- $r(a, B, C), s(C, D)$
$s(C, D)$	fb	$Q_2 : ans(D)$	:- $r(a, B, C), s(C, D)$
$p(D)$	f		

The two queries have the same subgoals but different heads, and both are not stable. However, we can still try to answer query Q_1 as follows: send a query $r(a, X, Y)$ to relation r. Assume we obtain three tuples: $\langle a, b_1, c_1\rangle$, $\langle a, b_2, c_2\rangle$, and $\langle a, b_2, c_3\rangle$. Thus, by retrieving these tuples, we know that the complete answer is a subset of $\{\langle b_1\rangle, \langle b_2\rangle\}$. Assume attributes A, B, C, and D have different domains. If relation p provides the tuples that allow us to retrieve some tuples $\langle c_1, d_1\rangle$ and $\langle c_2, d_2\rangle$ from s, we can know that $\{\langle b_1\rangle, \langle b_2\rangle\}$ *is* the complete answer. On the other hand, if relation p does not provide tuples that allow us to compute the answer $\{\langle b_1\rangle, \langle b_2\rangle\}$, we do not know whether we have computed the complete answer to Q_1.

We can try to answer query Q_2 in the similar way. After the first subgoal is processed, we also obtain the same three tuples from r. However, no matter what tuples are retrieved from relation s, we can never know the complete answer to Q_2, since there can always be a tuple $\langle c_1, d'\rangle$ in relation s that has not been retrieved, and d' is in the answer to Q_2. For both Q_1 and Q_2, if after processing the first subgoal we obtain no tuples from r that satisfy this subgoal, then we know that their complete answers are empty.

An important observation on these two queries is that Q_1's distinguished variable B can be bound by the answerable subgoal $r(a, B, C)$, while Q_2's distinguished variable D cannot be bound. Based on this observation, we develop a decision tree (Figure 1) that guides the planning process to compute the complete answer to a CQ. The shaded nodes in the figure are where we can conclude about whether we can compute the complete answer.

Now we explain the decision tree in details. Given a CQ Q and a database D, we first minimize a CQ Q by deleting its redundant subgoals, and compute its minimal equivalent Q_m (arc 1 in Figure 1). Then we test the feasibility of the query Q_m by calling the Inflationary algorithm; that is, we test whether Q_m has a feasible order of all its subgoals. If so (arc 2 in Figure 1), Q_m (thus Q) is stable, and its answer can be computed following a feasible order of all the subgoals in Q_m.

If Q_m is not feasible (arc 3), we compute all its answerable subgoals Φ_a, and check if all the *distinguished* variables are bound by the subgoals Φ_a. There are two cases:

1. If all the distinguished variables are bound by the subgoals Φ_a (arc 4), then the complete answer may be computed even if the supplementary relation [2, 24] (denoted I_a) of subgoals Φ_a is not empty. We compute the supplementary relation I_a of these subgoals following a feasible order of Φ_a.
   (a) If I_a is empty (arc 5), then we know that the complete answer is empty.
   (b) If I_a is not empty (arc 6), let I_a^P be the projection of I_a onto the distinguished variables. We use all the bindings from the query and the relations to retrieve as many tuples as possible. (See [8, 17] for the details.) Let Φ_{na} denote all the nonanswerable subgoals.
      i. If for every tuple $t^P \in I_a^P$, there is a tuple $t_a \in I_a$, such that the projection of t_a onto the distinguished variables is t^P, and t_a can join with some tuples for all the subgoals Φ_{na} (tuple t^P is called *satisfiable*), then we know that I_a^P is the complete answer to the query (arc 7).
      ii. Otherwise, the complete answer is not computable (arc 8).
2. If some distinguished variables are not bound by the subgoals Φ_a (arc 9), then the complete answer is not computable, unless the supplementary relation I_a is empty. Similarly to the case of arc 4, we compute I_a by following a feasible order of Φ_a. If I_a is empty (arc 10), then the complete answer is empty. Otherwise (arc 11), the complete answer is not computable.

While traversing the decision tree from the root to a leaf node, we may reach a node where we do not know whether the complete answer is computable until we traverse one level down the tree. Two planning strategies can be adopted at this kind of nodes: a pessimistic strategy and an optimistic strategy. A *pessimistic* strategy gives up traversing the tree once the complete answer is unlikely to be computable. On the contrary, an *optimistic* strategy is optimistic about the possibility of computing the complete answer, and it traverses one more level by taking the corresponding operations.

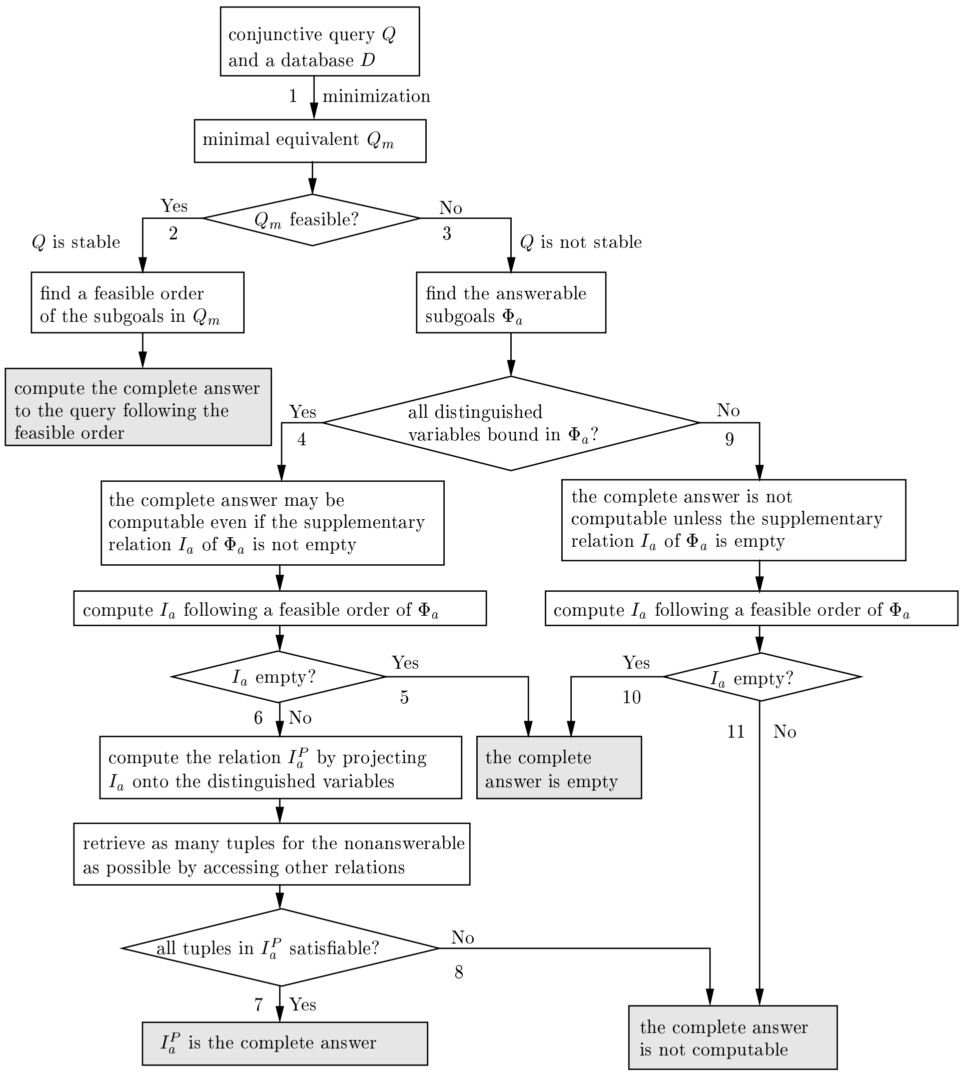

Fig. 1. Decision tree for computing the complete answer to a conjunctive query

6 Query Containment in the Presence of Binding Patterns

In the cases we cannot compute the complete answer to a query, we often want to compute the maximal answer to the query. In this section we study the following problem: *given two conjunctive queries on relations with limited access patterns, how to test whether the maximal answer to the first query is contained in the maximal answer to the second one?* We show this problem is decidable, and discuss how to test the query containment efficiently.

For a CQ Q on relations $\mathcal{R}$ with binding patterns, let $\Pi(Q, \mathcal{R})$ denote the program that computes the maximal answer to the query by using *only* the bindings from query Q and relations $\mathcal{R}$. It is shown in [8, 17] how the program $\Pi(Q, \mathcal{R})$ is constructed. $\Pi(Q, \mathcal{R})$ can be a recursive datalog program, even if the query Q itself is not recursive. That is, we might access the relations repeatedly to retrieve tuples, and use the new bindings in these tuples to retrieve more tuples from the relations. Formally, our containment problem is: *Given two conjunctive queries Q_1 and Q_2 on relations $\mathcal{R}$ with limited access patterns, how to test whether $\Pi(Q_1, \mathcal{R}) \sqsubseteq \Pi(Q_2, \mathcal{R})$?* The following theorem shows that this problem is decidable, even though containment of datalog programs is undecidable [23].

Theorem 8. *For two conjunctive queries Q_1 and Q_2 on relations $\mathcal{R}$ with binding patterns, whether $\Pi(Q_1, \mathcal{R}) \sqsubseteq \Pi(Q_2, \mathcal{R})$ is decidable.*

Proof. (Sketch) We can show that $\Pi(Q_1, \mathcal{R})$ and $\Pi(Q_2, \mathcal{R})$ are monadic datalog programs. A datalog program is *monadic* if all its recursive predicates [24] are monadic (i.e., with arity one); its nonrecursive IDB predicates can have arbitrary arity. Cosmadakis et al. [6] show that containment of monadic programs is decidable. Thus our containment problem is decidable.

This containment problem is recently studied in [19]. They prove the same decidability result using a different technique. There are some differences between our approach to the decidability result and their approach. The decidability proof in that paper is based on the assumption that the set of initial bindings for the contained query is a subset of the initial bindings for the containing query. We loosen this assumption because the decidability result holds even if the two queries have different initial bindings. Another difference between the two approaches is that we assume that the contained query is a conjunctive query, while in [19] the contained query can be a recursive datalog query. Finally, [19] uses the source-centric approach to information integration [25], which is different from the query-centric approach [25] that is taken in our framework. However, we can easily extend our technique [16] to the source-centric approach.

[6] involves a complex algorithm that uses tree-automata theory to test containment of monadic programs. If one of the two programs in the test is *bounded* (i.e., it is equivalent to a finite union of conjunctive query), then the containment can be tested more efficiently using the algorithms in [4, 5, 22]. Therefore, we are interested in how to test the boundedness of the program $\Pi(Q, \mathcal{R})$ for a query Q on relations $\mathcal{R}$ with binding patterns.

It is shown in [6] that boundedness is also decidable for monadic datalog programs, although it is not decidable in general [10]. However, testing boundedness of monadic programs also involves a complex algorithm [6]. In [16] we study this problem for the class of connection queries [17]. Informally, a connection query is a conjunctive query on relations whose schemas are subsets of some global attributes, and some values are given for certain attributes in the query. In [16] we give a polynomial-time algorithm for testing the boundedness of the datalog program for a connection query.

7 Related Work

Several works consider binding patterns in the context of answering queries using views [8, 14, 1]. Rajaraman, Sagiv, and Ullman [21] propose algorithms for answering queries using views with binding patterns. In that paper all solutions to a query compute the complete answer to the query; thus only stable queries are handled. Duschka and Levy [8] solve the same problem by translating source restrictions into recursive datalog rules to obtain the maximally-contained rewriting of a query, but the rewriting does not necessarily compute the query's complete answer. Li et al. [18] study the problem of generating an executable plan based on source restrictions. [9, 28] study query optimization in the presence of binding patterns. Yerneni et al. [27] consider how to compute mediator restrictions given source restrictions. These four studies do not minimize a conjunctive query before checking its feasibility. Thus, they regard the query Q_2 in Example 2 as an unsolvable query. In [17] we study how to compute the maximal answer to a conjunctive query with binding patterns by borrowing bindings from relations not in the query, but the computed answer may not be the complete answer. As we saw in Section 5, we can sometimes use the approach in that paper to compute the complete answer to a nonstable conjunctive query. Levy [13] considers the problem of obtaining complete answers from incomplete databases, and the author does not consider relations with binding restrictions.

Acknowledgments: We thank Foto Afrati, Mayank Bawa, Rada Chirkova, and Jeff Ullman for their valuable comments on this material.

References

1. F. N. Afrati, M. Gergatsoulis, and T. G. Kavalieros. Answering queries using materialized views with disjunctions. In *ICDT*, pages 435–452, 1999.
2. C. Beeri and R. Ramakrishnan. On the power of magic. In *PODS*, pages 269–283, 1987.
3. D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. Query answering using views for data integration over the Web. *WebDB*, pages 73–78, 1999.
4. A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. *STOC*, pages 77–90, 1977.
5. S. Chaudhuri and M. Y. Vardi. On the equivalence of recursive and nonrecursive datalog programs. In *PODS*, pages 55–66, 1992.

6. S. S. Cosmadakis, H. Gaifman, P. C. Kanellakis, and M. Y. Vardi. Decidable optimization problems for database logic programs. *STOC*, pages 477–490, 1988.
7. O. M. Duschka. Query planning and optimization in information integration. *Ph.D. Thesis, Computer Science Dept., Stanford Univ.*, 1997.
8. O. M. Duschka and A. Y. Levy. Recursive plans for information gathering. In *IJCAI*, 1997.
9. D. Florescu, A. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *SIGMOD*, pages 311–322, 1999.
10. H. Gaifman, H. G. Mairson, Y. Sagiv, and M. Y. Vardi. Undecidable optimization problems for database logic programs. *Journal of the ACM*, pages 683–713, 1993.
11. A. Gupta, Y. Sagiv, J. D. Ullman, and J. Widom. Constraint checking with partial information. In *PODS*, pages 45–55, 1994.
12. A. Klug. On conjunctive queries containing inequalities. *Journal of the ACM*, 35(1):146–160, January 1988.
13. A. Y. Levy. Obtaining complete answers from incomplete databases. In *Proc. of VLDB*, pages 402–412, 1996.
14. A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *PODS*, pages 95–104, 1995.
15. C. Li. Computing complete answers to queries in the presence of limited access patterns (extended version). *Technical report, Computer Science Dept., Stanford Univ.,* `http://dbpubs.stanford.edu:8090/pub/1999-11`, 1999.
16. C. Li and E. Chang. Testing query containment in the presence of limited access patterns. *Technical report, Computer Science Dept., Stanford Univ.,* `http://dbpubs.stanford.edu:8090/pub/1999-12`, 1999.
17. C. Li and E. Chang. Query planning with limited source capabilities. In *ICDE*, pages 401–412, 2000.
18. C. Li, R. Yerneni, V. Vassalos, H. Garcia-Molina, Y. Papakonstantinou, J. D. Ullman, and M. Valiveti. Capability based mediation in TSIMMIS. In *SIGMOD*, pages 564–566, 1998.
19. T. Millstein, A. Levy, and M. Friedman. Query containment for data integration systems. In *PODS*, 2000.
20. T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *Proc. of VLDB*, pages 122–133, 1998.
21. A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns. In *PODS*, pages 105–112, 1995.
22. Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM*, 27(4):633–655, 1980.
23. O. Shmueli. Equivalence of datalog queries is undecidable. *Journal of Logic Programming*, 15(3):231–241, 1993.
24. J. D. Ullman. *Principles of Database and Knowledge-base Systems, Volumes II: The New Technologies.* Computer Science Press, New York, 1989.
25. J. D. Ullman. Information integration using logical views. In *ICDT*, pages 19–40, 1997.
26. G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, 1992.
27. R. Yerneni, C. Li, H. Garcia-Molina, and J. D. Ullman. Computing capabilities of mediators. In *SIGMOD*, pages 443–454, 1999.
28. R. Yerneni, C. Li, J. D. Ullman, and H. Garcia-Molina. Optimizing large join queries in mediation systems. In *ICDT*, pages 348–364, 1999.
29. X. Zhang and M. Ozsoyoglu. On efficient reasoning with implication constraints. In *DOOD*, pages 236–252, 1993.