Processing First-Order Queries under Limited Access Patterns

Alan Nash
Department of Mathematics
University of California, San Diego
La Jolla, CA 92093-0112
anash@math.ucsd.edu

Bertram Ludäscher
San Diego Supercomputer Center
University of California, San Diego
La Jolla, CA 92093-0505
ludaesch@sdsc.edu

ABSTRACT

We study the problem of answering queries over sources with limited access patterns. Given a first-order query Q, the problem is to decide whether there is an equivalent query which can be executed observing the access patterns restrictions. If so, we say that Q is *feasible*. We define *feasible* for first-order queries—previous definitions handled only some existential cases—and characterize the complexity of many first-order query classes. For each of them, we show that deciding feasibility is as hard as deciding containment. Since feasibility is undecidable in many cases and hard to decide in some others, we also define an approximation to it which can be computed in **NP** for any first-order query and in **P** for unions of conjunctive queries with negation. Finally, we outline a practical overall strategy for processing first-order queries under limited access patterns.

1. INTRODUCTION

We study the problem of answering queries over sources with limited access patterns, i.e., where the source relations cannot be accessed in arbitrary ways, but only when certain combinations of attribute values are given. Consider a source relation *book*(*isbn*,*author*,*publisher*). The access patterns for *book* may prescribe that at least one of *isbn* or *author* must be given as input in order to access the relation, whereas only providing a value for *publisher* is not sufficient. This is specified by giving the access patterns $book^{\mathrm{ioo}}$, $book^{\mathrm{oio}}$, where 'i' denotes a required *input slot* and 'o' denotes an *output slot* for which no value is required.[1]

Given a query Q over a schema with limited access patterns, we are interested in the following questions:

- Is Q *executable* "as-is", i.e., in some execution order implied by the syntactic form of the query expression?
- Or is Q *feasible*, i.e., logically equivalent to an executable query Q'?

Executablility depends on the syntactic form of the query expression and is easy to check. In contrast, feasibility is a more "robust" semantic notion since it includes all equivalent query expressions, irrespective of their specific syntactic form.

Example 1 Consider two relations $pub(a, p)$ and $coa(a, a')$ which hold, respectively, if author a has published a book with publisher p and if a and a' have co-authored a book. Assume the only available access patterns are pub^{io} and coa^{io}. The query "*find all authors x who have published only where Ullman has*" can be expressed as $P(x) :=$

$$\forall y\,(\,pub(x, y) \rightarrow pub(\texttt{ullman}, y)\,)\ .$$

It turns out that the query $P(x)$ is neither executable nor feasible (intuitively, since we cannot bind x).

In contrast, the query "*find all co-authors x of Knuth who have published only where Ullman has*" is feasible. An equivalent executable query is $Q(x) :=$

$$coa(\texttt{knuth}, x) \wedge \forall y\,(\,pub(x, y)\ \rightarrow\ pub(\texttt{ullman}, y)\,)\ .$$

We can annotate each occurrence of an atom in Q with a legal access pattern in such a way, that the given input requirements are observed by the canonical execution order: after invoking $coa^{\mathrm{io}}(\texttt{knuth}, x)$, we obtain a binding for x, so that $pub^{\mathrm{io}}(x, y)$ is then observed. Also $pub^{\mathrm{io}}(\texttt{ullman}, y)$ is observed, since providing a value in an output slot (here: y) is allowed.

We can also express FO queries in a rule form which we call FO-datalog (resulting from a translation of FO into nonrecursive datalog with negation).

Example 2 (FO-datalog) The above query $Q(x)$ is expressed in rule form as follows:

$$\begin{array}{lcl} E(x) & \leftarrow & pub(x, y), \neg pub(\texttt{ullman}, y). \\ Q(x) & \leftarrow & coa(\texttt{knuth}, x), \neg E(x). \end{array}$$

Note that in the translation from FO to FO-datalog $\forall$ is replaced by $\neg\exists\neg$ and auxiliary IDB rules (here for E) may be introduced.

[1] Other authors use 'b' and 'f' (for bound and free) instead of 'i' and 'o' (for input and output), but we reserve the former for variables under or not under the scope of a quantifier, respectively.

PODS 2004 June 14-16, 2004, Paris, France.

A *query plan* is an executable query together with an annotation that satisfies the given access patterns and induced input requirements. For our $Q(x)$ above, a query plan is:

$$\begin{array}{lcl} E^{\mathrm{i}}(x) & \leftarrow & \mathit{pub}^{\mathrm{io}}(x,y), \neg \mathit{pub}^{\mathrm{io}}(\texttt{ullman}, y). \\ Q^{\mathrm{o}}(x) & \leftarrow & \mathit{coa}^{\mathrm{io}}(\texttt{knuth}, x), \neg E^{\mathrm{i}}(x). \end{array}$$

Note that the subgoal $\mathit{pub}^{\mathrm{io}}(x,y)$ requires x to be bound. This is achieved by annotating the head of the new rule for $E(x)$ with the pattern E^{i}, indicating that x is an *input parameter* of that rule. The occurrence $\neg E^{\mathrm{i}}(x)$ in the rule body is thus annotated accordingly, which is indeed valid since we can obtain a binding for x from the preceding subgoal $\mathit{coa}^{\mathrm{io}}(\texttt{knuth}, x)$.

The examples above involve universal queries (see Π_1 in the preliminaries). We show that deciding feasiblity is $\mathbf{\Pi}_2^{\mathrm{P}}$-complete for this class of queries. However, if we define the relations *pub* and *coa* above as conjunctive views over the *book* relation (e.g., $\mathit{pub}(a,p) \leftarrow \mathit{book}(i,a,p)$), we obtain query expressions of the form $\forall \bar{x} \exists \bar{y}\, F$, i.e., Π_2 formulas. We show that in general feasibility is undecidable for Π_2 formulas.

In order to support the most efficient query planning, it is important to handle each query expression according to its form. For example, if a formula is not in DNF, we can first convert it to DNF, then process it by an algorithm that handles DNF formulas. However, conversion to DNF may cause an exponential length increase. In particular, converting positive existential formulas to unions of conjunctive queries gives an upper bound of **NEXP** for deciding feasibility, whereas we obtain an upper bound of $\mathbf{\Pi}_2^{\mathrm{P}}$ which matches our lower bound. This is the motivation for studying a large number of query classes: by taking into account the form of the query, a query planner can choose an appropriate optimal strategy.

Contributions and Organization

We extend the notions of executable and feasible beyond the existential case and characterize the complexity of deciding feasibility for many first-order query classes. In particular, we provide new results for queries not in DNF and for universal queries. Furthermore, we define an approximation to feasibility which can be computed in **NP** for general first-order queries and in **P** for unions of conjunctive queries with negation.

In Section 2 we define access patterns and recall some notions including query containment. We also introduce a uniform notation for the FO fragments we study.

In Section 3 we characterize the complexity of deciding feasibility for several quantifier-free and existential FO fragments. We show that for these, feasibility is exactly as hard as query containment.

In Section 4 we extend the notion of executability to all of FO. We show that query feasibility is exactly as hard as query containment for Π_1 formulas, and is undecidable for Σ_k and Π_k formulas with $k \geqslant 2$.

Our results on the complexity of query feasibility establish that for all the query classes $\mathcal{L}$ we study, checking feasibility is as hard as checking containment. The following two tables summarize the complexity of deciding containment and feasibility.[2]

The previous results on feasibility are:

Σ_1^+C, Σ_1^+DNF	**NP**-complete	[RSU95], [LC01b]
Σ_1C, Σ_1DNF	$\mathbf{\Pi}_2^{\mathrm{P}}$-complete	[NL04]

The new results on feasibility, shown here are:

Σ_0^+C, Σ_0^+DNF, Σ_0C, Σ_0DNF	**P**
Σ_0^+, Σ_0	**coNP**-complete
Σ_1^+, Σ_1, Π_1	$\mathbf{\Pi}_2^{\mathrm{P}}$-complete
Σ_k, Π_k $\quad (k \geqslant 2)$	undecidable

For containment, we prove that the complexity for quantifier-free (Σ_0) and positive quantifier-free (Σ_0^+) formulas is **coNP**-complete. The remaining complexities are either widely known or easy to prove (see, e.g., [CM77] and [SY80]). As usual, we assume the schema is fixed.

For feasibility, the complexity for conjunctive queries (Σ_1^+C) was shown in [LC01b] and that for unions of conjunctive queries (Σ_1^+DNF) is implied by their work. In [NL04] we established the complexity for conjunctive queries with negation (Σ_1C) and unions of conjunctive queries with negation (Σ_1DNF). All other results on feasibility are new.

In Section 5 we present an intermediate notion, *orderability*, which is decidable for FO and aimed at practical approximations of feasibility, e.g., for query planning in mediator systems. We show that orderability is in **P** for Σ_1DNF and in **NP** for arbitrary FO formulas.

In Section 6 we show how our theoretical investigations lead to a practical overall strategy for processing first-order queries with access pattern restrictions.

Due to space restrictions, we have omitted many proofs and instead include them in the appendix.

Related Work

There has been a considerable amount of work in the area of query processing with limited query capabilities [Ull88, RSU95, LRO96, FLMS99, DGL00, MLF00, VP00, LC01b, Li03]. However, the complexity of query feasibility under access pattern restrictions has been largely open until now. The work by Li et al. [LC01b, Li03] is most closely related to ours. They show that for conjunctive queries (CQ) and conjunctive queries with union (UCQ), the notion of feasibility is closely related to that of *minimal queries* (and thus to containment). In particular, they show that (i) for CQ, deciding feasibility[3] is **NP**-complete, (ii) for $Q \in$ UCQ, $Q = Q_1 \vee \cdots \vee Q_n$ is feasible iff each minimal equivalent Q_i' of Q_i is orderable, and (iii) feasibility is undecidable for

[2]The preliminaries explain our uniform query class notation.
[3]they call it *stability*

datalog. They also provide an algorithm that in some cases can compute complete answers to CQ queries which are not feasible (the completeness depends on the given database instance).

In [NL04] we extend these results to conjunctive queries with negation ($\text{CQ}^\neg$) and unions of conjunctive queries with negation ($\text{UCQ}^\neg$) and show that for both classes deciding feasibility is $\mathbf{\Pi}_2^{\text{P}}$-complete. A uniform treatment for $\text{UCQ}^\neg$ and its subclasses CQ, $\text{CQ}^\neg$, and UCQ is obtained by the notion of the *answerable part* $\text{ans}(Q)$ of Q, which for those classes is shown to be the minimal feasible query containing Q (see Theorem 5 below). In [NL04] we also extend previous work [Li03] on the *runtime handling* of query answers and provide an elegant and efficient way to compute underestimates and overestimates of answers to infeasible queries, often allowing to quantify the degree of completeness at runtime.

In the context of data integration, several works have studied how to retrieve additional answers, e.g., by a "domain enumeration approach" that uses bindings from relations not necessarily mentioned in the query [DL97, LC01a]. It is also common to consider source capabilities expressed as conjunctive views with access patterns. An executable query plan in this setting is one which (i) expresses the query using the views only and (ii) observes the access patterns. [RSU95] show that for conjunctive queries and views finding an equivalent executable plan (i.e., deciding feasibility) is **NP**-complete. [DGL00] give a polynomial time algorithm for computing a *maximally-contained* (not necessarily equivalent) plan. The maximally-contained plan may be recursive (a datalog program) even if the query and source views are conjunctive. Summarizing, these works seek to improve "answerability" of infeasible queries including maximally-contained rewritings, while we characterize the complexity of query feasibility, i.e., equivalent rewritings. In particular, for unions of conjunctive queries with negation ($\text{UCQ}^\neg$), our notation of answerable part $\text{ans}(Q)$ provides a *minimal executable* and *minimal feasible* query *containing* Q.

As we will see, the notion of executability is related to that of *safety*. The latter can be see as a decidable syntactic approximation of domain independence (which is undecidable); see [GT91] for different classes of safe queries (e.g., *evaluable*, *allowed*, and *range-restricted*). Similarly, our syntactic notions of executability and orderability can be seen as approximations of the semantic notion of feasibility.

2. PRELIMINARIES

A *term* is a variable or constant. By $\bar{x}$ we denote a finite sequence of terms $x_1, \ldots, x_k$. FO *formulas* are defined in the usual way using predicate symbols, terms, and logic symbols $\forall, \exists, \wedge, \vee, \neg$. We use lowercase letters $x, y, \ldots$ for terms, $p, q, r, \ldots$ for predicate symbols, and uppercase letters $P, Q, \ldots$ for queries, IDB predicates, and programs. A formula with n free variables defines an n-ary query Q; $\text{free}(Q)$ denotes the *free* variables, $\text{vars}(Q)$ *all* variables (bound or free) of Q.

IDB predicates occur in the rule form (datalog) of FO. A *datalog rule* is an expression of the

$$P(\bar{z}) \leftarrow \ell_1(\bar{x}_1), \ldots, \ell_n(\bar{x}_n)$$

where each $\ell_i(\bar{x}_i)$ in the rule *body* is a *literal*, i.e., an atom $r(\bar{x})$ or its negation $\neg r(\bar{x})$. A datalog rule (with negation) corresponds to a conjunctive query (with negation) in the obvious way. Predicates in the rule *head*, here $P(\bar{z})$, are called *intensional* or *view predicates* (IDB) and denoted in uppercase; those occurring only in bodies are *extensional* or *base predicates* (EDB) and denoted in lowercase. A datalog *program* is a finite set of datalog rules.

Any FO formula F corresponds to a non-recursive datalog program P_F with negation and vice versa, using a canonical translation (cf. [GT91] and [AHV95]). We call the so-obtained class of programs *FO-datalog*. Since P_F is non-recursive, it has a unique (stratified) semantics. In the translation of F to P_F, auxiliary IDB predicates may be introduced, and universal quantification $\forall \bar{x}\, G$ is replaced by $\neg\exists\bar{x}\neg\, G$, which is then translated using auxiliary rules as shown in Example 2. It should be clear that in P_F each IDB predicate appears in the body of at most one rule and at most once. We call a program obtained by such translation a *FO-datalog* program. Notice that FO-datalog programs for Σ_1DNF queries have no auxiliary IDB predicates.

Definition 1 (Safe Programs) A FO-datalog program P is *safe* if every variable in a rule appears positively in the body of that rule.

Similar, but longer definitions can be given for safety of FO formulas (see [GT91] for different variations).

In addition to the well-known complexity classes **P**, **NP**, and **coNP**, we also refer to $\mathbf{\Pi}_2^{\text{P}} := \mathbf{coNP^{NP}}$, the class of languages decidable by co-nondeterministic machines with oracle access to an **NP**-complete problem. Equivalently, $\mathbf{\Pi}_2^{\text{P}}$ consists of all the languages $\mathcal{L}$ for which there is a language $\mathcal{L}' \in \mathbf{P}$ so that

$$x \in \mathcal{L} \text{ iff } \forall y \exists z\, \langle x, y, z\rangle \in \mathcal{L}'$$

where the lengths of y and z are bounded by some polynomial in the length of x. We write $\mathcal{L} \leqslant_m^{\text{P}} \mathcal{L}'$ for "$\mathcal{L}$ is polynomial-time many-one reducible to $\mathcal{L}'$." The complexity of all problems we discuss is in terms of the length of the formulas or programs that define the queries. Since the length of formulas and the corresponding FO-datalog programs are polynomially related, it is not necessary to distinguish between the two in the statement of the complexity results we present.

A formula is in *prenex normal form* (PNF) if it consists of a quantifier block followed by a quantifier-free formula. A formula is in *negation normal form* (NNF) if negation only occurs in front of atoms. A formula is in *disjunctive normal form* (DNF) if it is in PNF and the quantifier-free formula is a disjunction of conjunctions of literals.

We discuss many classes of queries, including some without well-established names. Therefore we introduce a new *uniform* notation, which departs somewhat from the existing notation for some classes.

We define classes of queries by the structure of the formulas that define them. We limit ourselves to formulas in PNF.

This is not a significant limitation, since arbitrary FO formulas can be converted into NNF and PNF efficiently with no significant change in length.

We refer to each PNF formula $Q \in \text{FO}$ by the structure of its quantifier block followed by some additional information. We write

- $Q \in \Sigma_0$ if Q has no quantifiers,
- $Q \in \Sigma_1$ if Q has only existential quantifiers, and
- $Q \in \Pi_1$ if Q has only universal quantifiers.

More generally, $Q \in \Sigma_k$ if the quantifier block of Q is of the form $\exists^\star\forall^\star\exists^\star\ldots$ with $k-1$ alternations of quantifiers. Similarly, $Q \in \Pi_k$ if the quantifier block of Q is of the form $\forall^\star\exists^\star\forall^\star\ldots$ with $k-1$ alternations of quantifiers in the sequence. We write

- $Q \in \Sigma_k^+$ if $Q \in \Sigma_k$ has no negations,
- $Q \in \Sigma_k\text{C}$ if $Q \in \Sigma_k$ has no disjunctions, and
- $Q \in \Sigma_k\text{DNF}$ if $Q \in \Sigma_k$ is in DNF;

and we handle Π_k similarly.

The following table shows the correspondence between our notation and other names for query classes.

Uniform	*a.k.a.*	*Description*
$\Sigma_1^+\text{C}$	CQ	Conjunctive queries
$\Sigma_1^+\text{DNF}$	UCQ	Unions of CQs
$\Sigma_1\text{C}$	$\text{CQ}^\neg$	CQs with negation
$\Sigma_1\text{DNF}$	$\text{UCQ}^\neg$	Unions of CQs with negation
Σ_1^+	$\exists\text{FO}^+$	Positive existential queries
Σ_1	$\exists$FO	Existential queries
Π_1	$\forall$FO	Universal queries
FO	FO	First-order queries

In all, we have six choices for each Σ_k. We characterize the complexity of the twelve cases for quantifier free and existential queries, as well as that of Π_1, and Π_k, Σ_k for $k \geqslant 2$.

Query P is said to be *contained* in query Q, denoted, $P \sqsubseteq Q$, if for every instance $\mathcal{D}$, $P(\mathcal{D}) \subseteq Q(\mathcal{D})$.

Definition 2 (Containment Problem)
CONT($\mathcal{L}$) is the following decision problem: Given $P, Q \in \mathcal{L}$ determine whether $P \sqsubseteq Q$ holds.

We prove the following result for quantifier-free formulas not in DNF, and use it in the proof of Theorem 8.

Theorem 1

- CONT(Σ_0^+) *is* **coNP**-*complete.*
- CONT(Σ_0) *is* **coNP**-*complete.*

Notice that, in contrast, CONT(Σ_0DNF) is in **P**.

Definition 3 (Access Pattern) An *access pattern* for a k-ary predicate symbol r is an expression of the form r^α where α is word of length k over the alphabet $\{\text{i}, \text{o}\}$. We call the jth position of r^α an *input slot* if $\alpha_j = \text{i}$ and an *output slot* if $\alpha_j = \text{o}$.

At runtime, we *must* provide values for input slots in order to execute the query, while for output slots such values are not required and instead can be retrieved from the source relation. In general, with access pattern r^α we may retrieve the set of tuples $\{\bar{y} \mid r(\bar{x}, \bar{y})\}$ as long as we supply the values of $\bar{x}$ corresponding to all input slots in r. We allow values to be supplied to output slots, but they are not required, that is "*bound is easier*" [Ull88].[4]

Definition 4 ($\mathcal{P}$-Annotation) Given a set $\mathcal{P}$ of access patterns, a $\mathcal{P}$-*annotation* on a query Q given by a FO formula or a FO-datalog program is an assignment of access patterns from $\mathcal{P}$ to each occurrence of a predicate symbol in Q.

Usually, we have in mind a fixed set $\mathcal{P}$ of access patterns so we do not refer to them explicitly.

3. THE EXISTENTIAL CASE

In this section, we analyze the complexity of feasibility of existential queries. We cover six classes of quantifier-free and six classes of existentially-quantified queries.

First, we define *executability* for Σ_1DNF queries. We then define *feasibility* in terms of executability. In the next section we generalize these definitions to all FO queries. It is easier to define the notion of an executable FO-datalog program than that of an executable formula, but it is more convenient to work with formulas in most of the proofs. For this reason, we provide both definitions. An executable program with the annotations that show its executability provides an *execution plan*: execute each rule separately (possibly in parallel) from left to right according to the access patterns annotations.

Definition 5 (Executable Program) A FO-datalog program $P \in \Sigma_1\text{DNF}$ is *executable* if it can be annotated so that every variable of a rule appears first (from left to right) positively in an output slot in the body of that rule.

Since $P \in \Sigma_1\text{DNF}$ is a disjunction $P_1 \vee \cdots \vee P_n$ of $\Sigma_1\text{C}$ queries, it is represented by n datalog rules (with negation) all having the same head. In particular, no new IDB predicates or auxiliary rules are introduced.

Definition 6 (Executable Formula) A formula $Q \in \Sigma_1\text{C}$ is *executable* if it can be annotated so that every variable of Q appears first (from left to right) in an output slot of a non-negated predicate. A formula $Q \in \Sigma_1\text{DNF}$ with $Q := Q_1 \vee \ldots \vee Q_k$ is *executable* if every Q_i is executable.

We consider the query which returns no tuples, written $Q(x_1, \ldots, x_n) \leftarrow \texttt{false}$ (or `false` for short), to be (vacuously) executable. In contrast, we consider the query with

[4]The justification for this common assumption is that if the data source does *not* accept a value, e.g., for y in $p^{\text{io}}(x, y)$, one can always ignore the binding and invoke $p(x, y')$ with y' unbound, and afterwards execute the join for $y' = y$.

an empty body, $Q(x_1, \ldots, x_n) \leftarrow \texttt{true}$ (or just `true`), to be non-executable: note that this query is safe iff the head has no variables. We may have both kinds of queries in ans(Q) defined below. From the definitions, it follows that executable queries are safe. The converse is false.

It is clear that a Σ_1DNF program is executable iff its corresponding formula is executable and vice versa. Checking executability of a query can be done in polynomial time, along the lines of the proof of Lemma 2 below.

Definition 7 ($\mathcal{L}$-Feasible) A query Q is *$\mathcal{L}$-feasible* if it is equivalent to an executable query $Q' \in \mathcal{L}$.

Definition 8 FEASIBLE($\mathcal{L}, \mathcal{L}'$) is the following decision problem: given a query $Q \in \mathcal{L}$, determine whether Q is $\mathcal{L}'$-feasible.

In the remainder of this section, whenever we say "feasible" we mean "Σ_1DNF-feasible."

The following two definitions and Lemma 2 are small modifications of those presented in [LC01b].

Definition 9 (Answerable Literal) Given a query $Q \in \Sigma_1$C, we say that a literal $\ell(\bar{x})$ (not necessarily in Q) is *Q-answerable* if there is an executable $Q^\ell \in \Sigma_1$C which is a conjunction $\ell(\bar{x})$ and some (not necessarily all) literals in Q.

Definition 10 (Answerable Part: ans(Q))
For $Q \in \Sigma_1$C and Q unsatisfiable, let ans(Q) := `false`. If Q is satisfiable, let ans(Q) be the conjunction of the Q-answerable literals in Q in the order specified by the algorithm in Lemma 2.[5] For $Q \in \Sigma_1$DNF with $Q = Q_1 \vee \ldots \vee Q_k$, let ans($Q$) := ans($Q_1$) $\vee \ldots \vee$ ans(Q_k).

Whether we think of ans(Q) as a formula or as a FO-datalog program, we consider ans(Q) to have the same arity as Q. In particular, if ans(Q) is a FO-datalog program we think of its rules as having the same heads as Q. Notice that for $Q \in \Sigma_1$DNF, it may be the case that for some i, ans(Q_i) has an empty body or a body in which not all of the variables in the head of Q appear. In this case, ans(Q) is not safe.

Lemma 2
If $Q \in \Sigma_1$DNF, we can compute ans(Q) *in quadratic time.*

Proof We consider the case when Q is given by a FO-datalog program and consists of a single rule. (Multiple rules are handled the same way, one at at time.) Give ans(Q) the same head as Q and build its body one subgoal at a time as follows. Start with $\mathcal{B}$, the set of bound variables, empty. Find the first subgoal $\ell(\bar{x})$ in Q not yet added to ans(Q) such that,

- $\ell(\bar{x})$ is positive and there is some access pattern for it in $\mathcal{P}$ such that all the input variables in $\ell(\bar{x})$ are in $\mathcal{B}$, or
- $\ell(\bar{x})$ is negative and its variables are in $\mathcal{B}$.

[5]Note that there is no actual circularity here.

If there is no such subgoal, stop. Otherwise, add $\ell(\bar{x}_i)$ to ans(Q), set $\mathcal{B} := \mathcal{B} \cup \{\bar{x}_i\}$, and repeat.

Clearly, this algorithm will add to ans(Q) all the Q-answerable literals in Q and no others.

The next two lemmas are easy to show; Theorem 5 is the main result of [NL04].

Lemma 3
If $Q \in \Sigma_1$DNF, then ans(Q) *is executable iff it is safe.*

Lemma 4 *If $Q \in \Sigma_1$DNF, then $Q \sqsubseteq$* ans(Q).

Theorem 5 *[NL04] If $Q, E \in \Sigma_1$DNF satisfy $Q \sqsubseteq E$ and E is executable then $Q \sqsubseteq$* ans(Q) $\sqsubseteq E$. *That is, if there is a minimal executable query containing Q, it is equivalent to* ans(Q).

Corollary 6 *$Q \in \Sigma_1$DNF is feasible iff* ans(Q) $\sqsubseteq Q$ *and* ans(Q) *is safe.*

Proof If Q is feasible, then there is an executable query $E \equiv Q$ and therefore ans(Q) $\equiv E$ by Theorem 5. Since E is executable, it must be safe and therefore ans(Q) is also safe. If ans(Q) $\sqsubseteq Q$, then by Lemma 4 we have $Q \equiv$ ans(Q). Since ans(Q) is safe, by Lemma 3, it is executable so Q is feasible.

Since checking safety of ans(Q) is computationally easy, in the remainder of this paper we concentrate on the test ans(Q) $\sqsubseteq Q$. First we give upper bounds, then we give lower bounds which are tight for all the subclasses of Σ_1 we consider.

Corollary 7 *If $\mathcal{L}$ is Σ_0^+C, Σ_0^+DNF, Σ_0C, Σ_0DNF, Σ_1^+C, Σ_1^+DNF, Σ_1C, or Σ_1DNF, then*

$$\text{FEASIBLE}(\mathcal{L}, \Sigma_1\text{DNF}) \leqslant_m^{\text{P}} \text{CONT}(\mathcal{L}).$$

Proof Given $Q \in \mathcal{L}$, compute ans(Q). If ans(Q) is not safe, then Q is not feasible. Otherwise, Q is feasible iff ans(Q) $\sqsubseteq Q$. The important point is that in all these cases, ans(Q) is no longer than Q.

If $\mathcal{L}$ is one of $\Sigma_0^+, \Sigma_0, \Sigma_1^+, \Sigma_1$, then ans($Q$) may be exponentially longer than Q, since to obtain ans(Q) we first need to convert Q to DNF. Therefore, we have to handle these cases separately.

Theorem 8

- FEASIBLE(Σ_0, Σ_1DNF) $\in$ **coNP**
- FEASIBLE(Σ_0^+, Σ_1DNF) $\in$ **coNP**
- FEASIBLE(Σ_1, Σ_1DNF) $\in \mathbf{\Pi}_2^{\text{P}}$
- FEASIBLE(Σ_1^+, Σ_1DNF) $\in \mathbf{\Pi}_2^{\text{P}}$

Theorem 9 *If $\mathcal{L}$ is Σ_0^+C, Σ_0^+DNF, Σ_0C, Σ_0DNF, Σ_0^+, Σ_0 Σ_1^+C, Σ_1^+DNF, Σ_1C, Σ_1DNF, Σ_1^+, or Σ_1, then*

$$\text{CONT}(\mathcal{L}) \leqslant_m^{\text{P}} \text{FEASIBLE}(\mathcal{L}, \Sigma_1\text{DNF}).$$

We summarize the results of this section in the following theorem.

Theorem 10 *If $\mathcal{L}$ is Σ_0^+C, Σ_0^+DNF, Σ_0^+, Σ_0C, Σ_0DNF, Σ_0 Σ_1^+C, Σ_1^+DNF, Σ_1^+ Σ_1C, Σ_1DNF, or Σ_1, then*

$$\text{CONT}(\mathcal{L}) \equiv_m^{\text{P}} \text{FEASIBLE}(\mathcal{L}, \Sigma_1\text{DNF}).$$

PROOF If $\mathcal{L}$ is Σ_0^+C, Σ_0^+DNF, Σ_0C, Σ_0DNF, Σ_1^+C, Σ_1^+DNF, Σ_1C, or Σ_1DNF, then the result follows directly from Theorem 9 and Corollary 7. For the cases of Σ_0^+, Σ_0, Σ_1^+, and Σ_1 cases, we have matching upper and lower bounds. We know that FEASIBLE(Σ_0^+, Σ_1DNF) and FEASIBLE(Σ_0, Σ_1DNF) are both **coNP**-complete. Since CONT(Σ_0^+) and CONT(Σ_0) are **coNP**-complete, we have FEASIBLE(Σ_0^+, Σ_1DNF) $\equiv_m^{\text{P}}$ CONT(Σ_0^+) and FEASIBLE(Σ_0, Σ_1DNF) $\equiv_m^{\text{P}}$ CONT(Σ_0). The case of Σ_1^+ and Σ_1 is similar.

4. THE UNIVERSAL AND FIRST-ORDER CASES

The restricted definitions of executable and feasible which we have introduced in the previous section apply only to existential queries. We now extend them for all FO queries. Again it is easier to define the notion of an executable FO-datalog program than that of an executable FO formula, but it is more convenient to work with the latter in many of the proofs.

For the extended definitions, we need to consider the case where values for some variables are already provided. That is, these variables are used as *parameters*. For this purpose, we now allow annotations on the heads of the rules of a program. The variables specified 'i' in the head are those which must be used as parameters when "invoking" the rule. Conversely, those specified 'o' in the head correspond to variables whose values the rule can generate. The annotation of an IDB predicate in the head must match its (single) occurrence in some body. In the case of formulas, we specify the variables which must be used as parameters in a set $\mathcal{V}$. The case we considered in the previous section is when $\mathcal{V} = \emptyset$ or, equivalently, when there are no parameter variables.

We will see that our extended definition of executability coincides with the restricted definition for Σ_1DNF. Furthermore, for Σ_1 queries, Σ_1DNF-feasibility coincides with Σ_1-feasibility.

The FO-datalog programs for Σ_1DNF queries in Section 3 contain a single (distinguished answer) IDB predicate. For arbitrary FO queries, new IDB predicates and rules are created, hence we allow annotations of those new rule heads. We call a variable which appears in the head of a rule in an input slot a *parameter (variable)*.

Definition 11 ($\mathcal{V}$-Executable FO Program) A FO-datalog program P is *executable* if it can be annotated so that every non-parameter variable of a rule appears first positively in an output slot in the body and so that the main head has input slots for the variables in $\mathcal{V}$.

Since we only consider FO-datalog programs in which every IDB appears in the body of at most one rule and at most once, we require a unique annotation on the head of an IDB rule which must also be used in the unique place where this IDB appears in the body of a rule.

It is clear that if a program is $\mathcal{V}$-executable then, given values for the variables in $\mathcal{V}$, it can be executed. Conversely, if a program can be executed given values for the variables in $\mathcal{V}$—without rewriting it and without reordering items within a rule—it is clear that it must be $\mathcal{V}$-executable.

Definition 12 ($\mathcal{V}$-Executable FO Formula) A formula Q is *$\mathcal{V}$-executable* if it can be annotated so that

- Q is atomic and all input-slot variables are in $\mathcal{V}$;
- Q is $\neg F$, F is $\mathcal{V}$-executable, and free(F) $\subseteq \mathcal{V}$;
- Q is $F \vee G$, F and G are $\mathcal{V}$-executable, and free(F) $\triangle$ free(G) $\subseteq \mathcal{V}$;[6]
- Q is $F \wedge G$, F is $\mathcal{V}$-executable, and G is $\mathcal{V} \cup$ free(F)-executable; or
- Q is $\exists x F$ and F is $\mathcal{V} \setminus \{x\}$-executable.

Similar to the existential case, one can show that a program P is $\mathcal{V}$-executable iff its corresponding formula is $\mathcal{V}$-executable and vice versa. In the case of formulas, we need to handle both disjunction and conjunctions, since if we were to rewrite one in terms of the other by using negation, we would get conditions on $\mathcal{V}$-executability more restrictive than the ones the definition above provides. On the other hand, it turns out that there is no meaningful way to provide special-case handling for the other binary logical connectives (e.g., $\rightarrow$, $\leftrightarrow$, $\not\leftrightarrow$).

We define feasible as before, but for all of FO.

Definition 13 ($\mathcal{V}$-Feasible) A query $Q \in$ FO is *$\mathcal{V}$-feasible* if it is equivalent to a $\mathcal{V}$-executable query Q'.

We say that a query is *executable* if it is $\emptyset$-executable and *feasible* if it is $\emptyset$-feasible. To simplify the presentation, in the rest of this section we only discuss $\emptyset$-feasibility. It is clear that the same complexity results hold for any $\mathcal{V}$.

Lemma 11 *If $Q \in \Sigma_1$ is $\mathcal{V}$-executable, then so is the negation normal form $Q' \equiv Q$ and the disjunctive normal form $Q' \equiv Q$.*

Lemma 11 implies Theorem 12, which shows that all the results of the previous section hold with FEASIBLE($\mathcal{L}, \Sigma_1$DNF) replaced by FEASIBLE($\mathcal{L}, \Sigma_1$).

Theorem 12

$$\text{FEASIBLE}(\Sigma_1, \Sigma_1) = \text{FEASIBLE}(\Sigma_1, \Sigma_1\text{DNF}).$$

In fact, given the following lemma, we can get the somewhat stronger result that Σ_1DNF-feasibility is the same as $\mathcal{L}$-feasibility where $\mathcal{L}$ is the class of formulas which, once converted to NNF, have only existential quantifiers.

[6] $F \triangle G := F \cup G \setminus F \cap G = (F \setminus G) \cup (G \setminus F)$ is the symmetric set difference of F and G.

Lemma 13 *If Q is a $\mathcal{V}$-executable NNF formula which has only existential quantifiers, then the formula Q' obtained by renaming variables so that each quantifier quantifies a different variable, then moving all the quantifiers to the front is also $\mathcal{V}$-executable.*

Notice that the definition of $\mathcal{V}$-executable is similar to the one for $\mathcal{V}$-safe, which we give for programs only.

Definition 14 ($\mathcal{V}$-Safe Programs) A FO-datalog program P is *$\mathcal{V}$-safe* if it can be annotated so that every non-parameter variable of a rule appears positively in the body and so that the main head has input slots for the variables in $\mathcal{V}$.

The following lemma follows immediately from the definitions. Its converse is false.

Lemma 14
If a program P is $\mathcal{V}$-executable, then P is $\mathcal{V}$-safe.

Lemma 15 *If a program P is $\mathcal{V}$-safe and all EDB predicates have all-output access patterns, then P is $\mathcal{V}$-feasible.*

PROOF To get P', reorder each rule of P so that all positive literals appear first. Annotate the heads of P' as P can be annotated to show that it is safe and annotate all the predicate symbols in P' with all-output access patterns. Then the rules of P and P' have the same variables. Since every non-parameter variable of a rule in P appears positively in the body of that rule, it appears first positively in an output slot of P'.

Theorem 16

- *For $k \geqslant 1$,* $\text{CONT}(\Sigma_k) \leqslant_m^{\text{P}} \text{FEASIBLE}(\Sigma_k, \text{FO})$.
- *For $k \geqslant 1$,* $\text{CONT}(\Pi_k) \leqslant_m^{\text{P}} \text{FEASIBLE}(\Pi_k, \text{FO})$.

Theorem 17 $\text{FEASIBLE}(\Pi_1, \text{FO}) \in \mathbf{\Pi}_2^{\text{P}}$.

We summarize the results of this section in the following theorem.

Theorem 18 $\text{FEASIBLE}(\Pi_1, \text{FO}) \equiv_m^{\text{P}} \text{CONT}(\Pi_1)$ *and, for $k \geqslant 2$,* $\text{FEASIBLE}(\Sigma_k, \text{FO})$ *and* $\text{FEASIBLE}(\Pi_k)$ *are undecidable.*

PROOF The first part follows from Theorems 16 and 17 since $\text{CONT}(\Pi_1)$ is $\mathbf{\Pi}_2^{\text{P}}$-complete. The second part follows from Theorem 16 since $\text{CONT}(\Sigma_2)$ and $\text{CONT}(\Pi_2)$ are undecidable.

5. ORDERABILITY

We have seen that testing feasibility of a query can be hard to decide or even undecidable. Therefore, it seems worthwhile to explore how to efficiently approximate feasibility. An obvious notion intermediate between executable and feasible is the following.

Definition 15 ($\mathcal{V}$-Orderable Program) A program P in FO-datalog is *orderable* if it can be annotated and the subgoals in the rules can be reordered so that every non-parameter variable of a rule appears first positively in an output slot in the body of that rule and so that the main head has input slots for the variables in $\mathcal{V}$.

It is clear that if a query is $\mathcal{V}$-executable, then it is $\mathcal{V}$-orderable.

Definition 16 ORDERABLE($\mathcal{L}$) is the following decision problem: Given a query $Q \in \mathcal{L}$, determine whether Q is orderable.

Theorem 19 ORDERABLE(FO) *is* **NP**-*complete.*

PROOF (sketch) ORDERABLE(FO) $\in$ **NP**: Guess an annotation for each IDB predicate, then check whether each rule is orderable as in the proof of Lemma 2, except initialize $\mathcal{B}$ to the set of parameters for that rule and check that no subgoals are left over.

To show that ORDERABLE(FO) is **NP**-hard, we reduce 3SAT to ORDERABLE(FO) as follows. Start with binary EDB predicate A with access patterns A^{io} and A^{oi} and unary EDB predicate B with access pattern B^{o}. Given a propositional 3CNF formula ψ, create a FO-datalog program Q_ψ with one rule Q_C for each clause C in ψ as follows. For each variable v in C, invent a new binary IDB predicate V. If v appears positively in C, add $V(x, y)$ to Q_C; otherwise add $V(y, x)$ to Q_C. Finally, add $B(x)$ to Q_C. For example, given the clause $C := u \vee v \vee \neg w$, we get the rules $Q_C(x, y) \leftarrow U(x, y), V(x, y), W(y, x), B(x)$. Finally, for every EDB U which appears in Q_C, add a rule of the form $B(x, y) \leftarrow A(x, y)$ to Q_ψ. The program Q_ψ is the disjunction of all these rules, where we interpret all the heads Q_C as the same head. If Q_ψ is orderable, there must be an annotation such that each EDB U corresponding to a variable u in ψ has exactly one 'o'. Out of such annotations we extract an assignment α satisfying ψ. If the annotation is U^{io}, the assignment sets u to true; otherwise, it sets u to false. Conversely, if ψ is satisfiable, we select an assignment α satisfying ψ and annotate each EDB U corresponding to a variable u in ψ with io if α sets u to true and with oi otherwise. This annotation witnesses that Q_ψ is orderable.

Notice that the proof of Theorem 19 actually shows that ORDERABLE(Σ_1^+) is **NP**-hard. On the other hand, note that ORDERABLE(Σ_1DNF) $\in$ **P** by Lemma 2 since a query $Q \in \Sigma_1$DNF is orderable iff it is equal to ans(Q) up to order of the subgoals.

Finally, notice that by Lemma 11 we can always push negation in before checking orderability. After we have pushed negation in, we can also push existential quantifiers out by Lemma 13 assuming there are no universal quantifiers. Unfortunately, pushing universal quantifers out in general reduces executability so once we have mixed quantifiers, there is no simple optimization that can be applied. In general, we get more executability by pushing existential quantifiers out and universal quantifiers in.

6. PUTTING IT ALL TOGETHER

Given our results from the previous sections, we now present an overall strategy that is aimed at efficiently determining whether a query $Q \in$ FO is feasible. Note that previous work was restricted to much more limited cases, i.e., approximations for Σ_1^+DNF [Li03] and Σ_1DNF [NL04].

1. Convert Q to NNF. This can be done efficiently, may

increase orderability, and allows to determine whether we fall in the existential, universal, or neither case.

2. Check whether Q is orderable. If so, then Q is feasible.

3. Check whether Q has only existential quantifiers. If so, move them to the front.

   (a) If Q is now in DNF, compute ans(Q). Check whether ans(Q) is safe and ans(Q) $\sqsubseteq Q$. If so, Q is feasible; otherwise, it is not.

   (b) Otherwise, apply the parallelizable algorithms outlined in the proof of Theorem 8.

4. Check whether Q has only universal quantifiers. If so, apply the parallelizable algorithm outlined in the proof of Theorem 17.

5. Apply some limited rewritings of Q and check again for orderability.

6. Give up.

7. DISCUSSION AND CONCLUSIONS

We have characterized the complexity of query feasibility in the presence of access patterns for a large number of classes. Our results extend previous ones and indicate that feasibility is just as hard as containment. However, we had to show this laboriously case by case, since there is no obvious uniform reduction. From a theoretician's point of view, we would like to find one, but we conjecture it does not exist. Also we are interested in investigating the problem of feasibility in the presence of integrity constraints.

Query processing in the presence of limited access patterns is even more important from a practicioner's point of view. Traditionally it is encountered in data integration, specifically over web sources. With the emergence of web services [WSD03] as a simple means for distributed computation and data access, the need to understand executability and feasibility of queries is further increased, since it is a fundamental problem, e.g., in web service composition [TKL03] and planning for distributed scientific workflows [DBG$^+$03, LAG03, HBCS03]. Clearly, our results belong to the realm of database theory since we characterize the inherent complexity of query feasibility – a prerequesite for rewriting queries into equivalent executable ones. However, we would like to emphasize that our work was motivated and is driven by a number of very practical engineering problems: In the Biomedical Informatics Research Network [BIR01], we are developing a database mediator system for federating heterogeneous brain data [GLM03, LGM03]. The current prototype takes a query against a global-as-view definition and unfolds it into a Σ_1DNF plan. We have used the notions of *answerable part* and *orderable*, as well as novel runtime techniques [NL04] in the prototype system. Our theoretical investigations have solved several of the pending algorithmic issues and we can now proceed with the extension of the mediator planner. In the BIRN project, as in many other projects pertaining to scientific data management, sources with limited query capabilities are ubiquitous. See Example 3 for an application example in the context of scientific workflows (a.k.a. "analysis pipelines").

Acknowledgements

We thank Alin Deutsch for valuable discussions on query containment, in particular, the proof of Theorem 1 was done jointly with him.

This work was supported by NSF/ITR grant 0225676 (SEEK), NIH 8P41 RR08605-08S1 (BIRN-CC), NSF/ITR 0225673 (GEON), and DOE/SciDAC DE-FC02-01ER25486 (SDM).

8. REFERENCES

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.

[BIR01] Biomedical Informatics Research Network Coordinating Center (BIRN-CC), University of California, San Diego. `http://nbirn.net/`, 2001.

[CM77] A. K. Chandra and P. M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *ACM Symposium on Theory of Computing (STOC)*, pp. 77–90, 1977.

[DBG$^+$03] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, R. Cavanaugh, and S. Koranda. Mapping Abstract Complex Workflows onto Grid Environments. *Journal of Grid Computing*, 1(1):25–39, 2003.

[DGL00] O. M. Duschka, M. R. Genesereth, and A. Y. Levy. Recursive Query Plans for Data Integration. *Journal of Logic Programming*, 43(1):49–73, 2000.

[DL97] O. M. Duschka and A. Y. Levy. Recursive Plans for Information Gathering. In *Proc. IJCAI*, Nagoya, Japan, 1997.

[FLMS99] D. Florescu, A. Y. Levy, I. Manolescu, and D. Suciu. Query Optimization in the Presence of Limited Access Patterns. In *SIGMOD*, pp. 311–322, 1999.

[GLM03] A. Gupta, B. Ludäscher, and M. Martone. BIRN-M: A Semantic Mediator for Solving Real-World Neuroscience Problems. In *ACM Intl. Conference on Management of Data (SIGMOD)*, 2003.

[GT91] 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.

[HBCS03] R. Hull, M. Benedikt, V. Christophides, and J. Su. E-Services: A Look Behind the Curtain. In *ACM Symposium on Principles of Database Systems (PODS)*, pp. 1–14, 2003.

[LAG03] B. Ludäscher, I. Altintas, and A. Gupta. Compiling Abstract Scientific Workflows into Web Service Workflows. In *15th Intl. Conference on Scientific and Statistical Database Management (SSDBM)*, Boston, Massachussets, 2003.

[LC01a] C. Li and E. Chang. Answering Queries with Useful Bindings. *ACM Transactions on Database Systems (TODS)*, 26(3), 2001.

[LC01b] C. Li and E. Y. Chang. On Answering Queries in the Presence of Limited Access Patterns. In *Intl. Conference on Database Theory (ICDT)*, 2001.

[LGM03] B. Ludäscher, A. Gupta, and M. E. Martone. *Bioinformatics: Managing Scientific Data*, chapter A Model-Based Mediator System for Scientific Data Management. Morgan Kaufmann, 2003.

[Li03] C. Li. Computing Complete Answers to Queries in the Presence of Limited Access Patterns. *Journal of VLDB*, 12:211–227, 2003.

[LP95] E. A. Lee and T. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–799, 1995.

[LRO96] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *Proceedings of the Twenty-second International Conference on Very Large Databases*, pp. 251–262, Bombay, India, 1996. VLDB Endowment, Saratoga, Calif.

[Lud03] B. Ludäscher. Kepler: Scientific Workflows Based on Dataflow Process Networks. In *e-Science Workflow Services*, National e-Science Center, Edinburgh, UK, 2003.

[MLF00] T. D. Millstein, A. Y. Levy, and M. Friedman. Query Containment for Data Integration Systems. In *Symposium on Principles of Database Systems*, pp. 67–75, 2000.

[NL04] A. Nash and B. Ludäscher. Processing Unions of Conjunctive Queries with Negation under Limited Access Patterns. In *Intl. Conference on Extending Database Technology (EDBT)*, Heraklion, Crete, Greece, 2004.

[RSU95] A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering Queries Using Templates with Binding Patterns. In *ACM Symposium on Principles of Database Systems (PODS)*, pp. 105–112, 1995.

[SDM01] Scientific Data Management Center (SDM). `http://sdm.lbl.gov/sdmcenter/`, 2001.

[SEE02] Science Environment for Ecological Knowledge (SEEK). `http://seek.ecoinformatics.org/`, 2002.

[SY80] Y. Sagiv and M. Yannakakis. Equivalences Among Relational Expressions with the Union and Difference Operators. *Journal of the ACM*, 27(4):633–655, 1980.

[TKL03] S. Thakkar, C. A. Knoblock, and J. Luis. A View Integration Approach to Dynamic Composition of Web Services. In *ICAPS 2003 Workshop on Planning for Web Services*, Trento, Italy, June 2003.

[Ull88] J. Ullman. The Complexity of Ordering Subgoals. In *ACM Symposium on Principles of Database Systems (PODS)*, 1988.

[VP00] V. Vassalos and Y. Papakonstantinou. Expressive Capabilities Description Languages and Query Rewriting Algorithms. *Journal of Logic Programming*, 43(1):75–122, 2000.

[WSD03] Web Services Description Language (WSDL) Version 1.2. `http://www.w3.org/TR/wsdl12`, June 2003.

APPENDIX

A. SCIENTIFIC APPLICATION EXAMPLE

In the introduction we used the usual example domain of books, publications etc. The following is a variant of Example 1, illustrating that similar queries also occur in scientific applications such as those encountered in several of our scientific data integration projects [BIR01, SEE02, SDM01].

Example 3 Consider the relations $oav(oid, attr, val)$ and $ok(attr, val)$ which hold, respectively, if the *attr* value of some data object *oid* is *val*, and if the value *val* for some *attr* is within an "ok-range" (e.g., above some threshold). We are interested in a "filter query" $F(x)$ which returns those objects x for which *all* of its attribute values are within the ok-range.[7] This can be expressed as follows:

$$F(x) := \quad \forall a \forall v \, (\, oav(x, a, v) \rightarrow ok(a, v) \,) \; .$$

The binary "*ok* relation" is really implemented as a function $ok : Attr \times Val \rightarrow Bool$. This is captured by the access pattern ok^{ii} since both arguments have to be provided. Similarly, the ternary object base relation *oav* can be accessed only using the pattern oav^{ioo}, i.e., at least the object-id has to be given. It is easy to see that $F(x)$ is *not feasible* since x cannot be bound. When represented in FO-datalog, the distinguished answer predicate can be annotated as $F^{\mathrm{i}}(x)$, indicating that x is a required input parameter. Thus, given any "enumerator query" $E^{\mathrm{o}}(x)$ for x values, an executable query plan for $E^{\mathrm{o}}(x) \wedge F^{\mathrm{i}}(x)$ can be obtained easily.

B. PROOFS

Definition 17 (Consistent *F*-Expansion) If $C \in \Sigma_0\mathrm{C}$, then we say that C' is a *consistent F-expansion* of C if C' is satisfiable, contains all literals from C, and consists only of literals formed from variables and predicate symbols in F, and every conjunction of C' with a literal formed from variables and predicate symbols in F that is not already in C' is unsatisfiable. In particular, if C is unsatisfiable, it has no consistent F-expansion for any F.

Definition 18 (exp(*P*)) Given $P \in \Sigma_0\mathrm{DNF}$ of the form $P := \bigvee_{1 \leqslant i \leqslant n} P_i$, define $\exp(P)$ to consist of the disjunction of all consistent P-expansions of each P_i. Given $P \in \Sigma_1\mathrm{DNF}$ of the form $\exists \bar{y} P'$ where $P' \in \Sigma_0\mathrm{DNF}$, define $\exp(P) := \exists \bar{y} \exp(P')$.

Clearly $\exp(P) \equiv P$.

Definition 19 (Containment Mapping) For $P, Q \in \Sigma_1\mathrm{C}$, a function σ: $\mathrm{vars}(Q) \rightarrow \mathrm{vars}(P)$ is a *containment mapping* if P and Q have the same free variables, σ is the identity on the free variables of Q, and, for every literal $\ell(\bar{y})$ in Q, the literal $\ell(\sigma \bar{y})$ is in P.

The following is easy to show (in fact, it shows that over a fixed schema $\mathrm{CONT}(\Sigma_1\mathrm{DNF}) \in \mathbf{\Pi}_2^{\mathrm{P}}$).

[7]Such filters often occur in larger scientific *analysis pipelines* [SEE02] or *scientific workflows* [Lud03], which are essentially dataflow process networks [LP95]. As part of such a network, the process implementing F has a single input port through which x tokens flow and two output ports $ok(x)$ and $bad(x)$, representing those tokens for which $ok(x)$ and $\neg ok(x)$ holds, respectively.

Theorem 20 *[SY80] If $P, Q \in \Sigma_1\mathrm{DNF}$ have the same predicate symbols and satisfy $\exp(P) := \bigvee_{1 \leqslant i \leqslant n} P_i$ and $Q := \bigvee_{1 \leqslant j \leqslant m} Q_i$ such that $P_i, Q_j \in \Sigma_1\mathrm{C}$ then $P \sqsubseteq Q$ iff for every i there is j and a containment mapping $Q_j \rightarrow P_i$.*

The requirement that P and Q have the same predicate symbols is to simplify the exposition. Otherwise, we would need $\exp(P)$ to include expansions with predicate symbols from both P and Q.

PROOF (sketch) If for some i, there is no such j, then the frozen instance $[P_i]$ is a counterexample to the containment. Otherwise, since every tuple in every database $\mathcal{D}$ must satisfy one of the P_is, it must also some Q_j and therefore $P \sqsubseteq Q$. Over a fixed schema of total arity k, every P_i is of length $O(n^k)$ where n is the number of variables in P so the test can be done in $\mathbf{\Pi}_2^{\mathrm{P}}$.

PROOF (**Theorem 1**) We know validity of propositional formulas is **coNP**-complete. Given a propositional formula G, we map it to a Σ_0^+ query Q_G as follows. Send every literal of the form r to $T(r)$ and every literal of the form $\neg r$ to $F(r)$. Otherwise keep the formula as it is. Define

$$P_G := \bigwedge_{r \in \mathrm{vars}(G)} (T(r) \vee F(r))$$

where the conjunction is over all propositional variables r appearing in G. Then

$$G \text{ is valid iff } P_G \sqsubseteq Q_G.$$

If G is not valid, then we can convert an assignment σ of propositional variables to $\{\mathtt{true}, \mathtt{false}\}$ which makes G false, into an assignment σ' given by

$$\sigma'(r) := \begin{cases} 0 & \text{if } \neg\sigma(r) \\ 1 & \text{if } \sigma(r) \end{cases}$$

over the database $\mathcal{D}$ of two elements 0 and 1, with unary relations $T^{\mathcal{D}} = \{1\}$ and $F^{\mathcal{D}} = \{0\}$. Clearly,

$$\mathcal{D} \models P_G \left[\sigma'\right] \text{ and } \mathcal{D} \not\models Q_G \left[\sigma'\right].$$

Conversely, given any $\mathcal{D}$ and σ' satisfying the condition above, we can define

$$\sigma(r) := \begin{cases} 0 & \text{if } \mathcal{D} \models F(r) \, [\sigma] \\ 1 & \text{otherwise.} \end{cases}$$

Clearly, σ makes G false.

On the other hand, given formulas $P, Q \in \Sigma_0$, we can check that $P \sqsubseteq Q$ as follows. For every conjunction C of literals in P and Q, check that $C \sqsubseteq P$ implies $C \sqsubseteq Q$. Then (and only then)

$$P \equiv \bigvee_{C \sqsubseteq Q} C \sqsubseteq Q.$$

Therefore, we can test whether $P \sqsubseteq Q$ with a co-nondeterministic Turing machine that has access to an oracle for containment of $\Sigma_0\mathrm{C}$ queries within Σ_0 queries. Since the latter containment can be done in polynomial time—essentially it just requires the evaluation of Q on the frozen instance of C—we can do this in **coNP**.

PROOF (**Theorem 8**) We can check feasibility of $Q \in \Sigma_0$ by working on the DNF equivalent Q' given by $Q' := \bigvee_{P \sqsubseteq Q} P$ where each P is a conjunction of literals in Q. Since we can not afford to compute Q' explicitly—doing so may cause an exponential length increase—we work in parallel on each disjunct P of Q'.

To check that $\text{ans}(Q') \sqsubseteq Q$, for every conjunction P of literals in Q, check that $P \sqsubseteq Q$ implies $\text{ans}(P) \sqsubseteq Q$. Then (and only then)

$$\text{ans}(Q') = \bigvee_{P \sqsubseteq Q} \text{ans}(P) \sqsubseteq Q.$$

That is, $\text{ans}(Q') \sqsubseteq Q$. Therefore, we can test whether $\text{ans}(Q') \sqsubseteq Q$ with a co-nondeterministic Turing machine that has access to an oracle for containment of Σ_0C queries within Σ_0 queries. Since $\text{CONT}(\Sigma_0) \in \mathbf{coNP}$, we can do this in $\mathbf{coNP}^{\mathbf{coNP}} = \mathbf{coNP}$.

To check that $\text{ans}(Q')$ is safe, for every conjunction P of literals in Q, check that if $P \sqsubseteq Q$, then $\text{ans}(P)$ is safe. Since $\text{CONT}(\Sigma_0) \in \mathbf{coNP}$ and since checking that $\text{ans}(P)$ can be done in $\mathbf{P}$, we can do this in $\mathbf{coNP}^{\mathbf{coNP}} = \mathbf{coNP}$.

We can check feasibility of $Q \in \Sigma_1$ as follows. Assume all existential quantifiers are in front (if not, we can efficiently move them to the front). Proceed on the quantifier free part of Q essentially as above, by working in parallel on the DNF equivalent Q'. The check that $\text{ans}(Q')$ is safe can be done in **coNP**. To check that $\text{ans}(Q') \sqsubseteq Q$, for every conjunction P' of literals in Q, check that $P \sqsubseteq Q$ implies $\text{ans}(P) \sqsubseteq Q$ where P is P' quantified as in Q. We can do this with a co-nondeterministic Turing machine that has access to an oracle for containment of Σ_1C queries within Σ_1 queries. Since $\text{CONT}(\Sigma_1) \in \mathbf{\Pi}_2^{\text{P}}$,

$$\text{FEASIBLE}(\Sigma_1, \Sigma_1\text{DNF}) \in \mathbf{coNP}^{\mathbf{\Pi}_2^{\text{P}}} = \mathbf{\Pi}_2^{\text{P}}.$$

PROOF (**Theorem 9**) Given $P \in \mathcal{L}$, define P' by replacing every atom of the form $r(\bar{x})$ with an atom of the form $r'(u\bar{x})$ where r' has arity one more than r and binding pattern $r'^{\text{iooo}\cdots}$. Similarly, given $Q \in \mathcal{L}$, define Q' by replacing every atom of the form $r(\bar{x})$ with an atom of the form $r'(v\bar{x})$ where r' has arity one more than r and binding pattern $r'^{\text{iooo}\cdots}$. Set

$$L'(\bar{x}) := \exists u(s(u) \wedge P'(\bar{x}) \wedge Q'(\bar{x}))$$

where s has binding pattern s^{o} and rearrange L' to obtain $L \in \mathcal{L}$ so that $L \equiv L'$ (for example, by putting L' in DNF). Then $P \sqsubseteq Q$ iff

$$P \sqsubseteq P \wedge Q \text{ iff } P' \sqsubseteq P' \wedge Q' \text{ iff } \text{ans}(L) \sqsubseteq L.$$

The first "iff" is immediate. The third "iff" follows from the fact that only (and exactly) those atoms that have the variable u appear in $\text{ans}(L)$ and so $\text{ans}(L) \equiv \exists u(s(u) \wedge P')$. The second "iff" can be verified as follows. If there is a database $\mathcal{D}$ and a tuple $\bar{a}$ so that $\mathcal{D} \models P[\bar{a}]$ and $\mathcal{D} \not\models P \wedge Q[\bar{a}]$ then we can obtain a database $\mathcal{D}'$ and a tuple $\bar{a}'$ so that $\mathcal{D} \models P'[\bar{a}']$ and $\mathcal{D}' \not\models P \wedge Q[\bar{a}']$ by replacing every relation r in $\mathcal{D}$ with a relation r' given by setting

$$r'(\bar{x}') \text{ iff } x'_1 = c \text{ and } r(x'_2 \ldots x'_{k+1})$$

for some constant c. Conversely, given $\mathcal{D}'$ and $\bar{a}'$ witnessing $P' \not\sqsubseteq P' \wedge Q'$, we obtain $\mathcal{D}$ and $\bar{a}$ witnessing $P \not\sqsubseteq P \wedge Q$ by replacing every relation r' in $\mathcal{D}'$ with a relation r given by projecting away the first column. That is,

$$r(x'_2 \ldots x'_{k+1}) \text{ iff } r'(\bar{x}').$$

PROOF (**Lemma 11**) If Q is $\neg(F \wedge G)$ and is $\mathcal{V}$-executable, then $\text{free}(F), \text{free}(G) \subseteq \mathcal{V}$ and F is $\mathcal{V}$-executable and G is $\mathcal{V} \cup \text{free}(F)$-executable, which implies that G is $\mathcal{V}$-executable. Therefore, $Q' := \neg F \vee \neg G$ is $\mathcal{V}$-executable.

If Q is $\neg(F \vee G)$ and is $\mathcal{V}$-executable, then $\text{free}(F), \text{free}(G) \subseteq \mathcal{V}$ and F, G are $\mathcal{V}$-executable. Therefore, $Q' := \neg F \wedge \neg G$ is $\mathcal{V}$-executable.

If Q is $\neg\neg F$ and is $\mathcal{V}$-executable, then certainly $Q' := F$ is $\mathcal{V}$-executable.

From above we can assume that Q is in negation normal form. We have to show that if

$$P = (F \vee G) \wedge (H \vee J)$$

is $\mathcal{V}$-executable, then

$$P' = (F \wedge H) \vee (F \wedge J) \vee (G \wedge H) \vee (G \wedge J)$$

is $\mathcal{V}$-executable. If P is $\mathcal{V}$-executable, then $F \vee G$ is $\mathcal{V}$-executable and $H \vee J$ is $\mathcal{V}'$-executable where $\mathcal{V}' := (V \cup \text{free}(F) \cup \text{free}(G))$. Then F, G are $\mathcal{V}$-executable and H, J are $\mathcal{V}'$-executable. This shows that each of $F \wedge H$, $F \wedge J$, $G \wedge H$, and $G \wedge J$ are $\mathcal{V}$-executable. Furthermore, we have $\text{free}(F) \triangle \text{free}(G) \subseteq V$ and $\text{free}(H) \triangle \text{free}(J) \subseteq \mathcal{V}'$ which implies

$$\text{free}(H) \triangle \text{free}(J) \subseteq \mathcal{V} \cup \text{free}(F), \mathcal{V} \cup \text{free}(G).$$

Therefore, $\text{free}(F \wedge H) \triangle \text{free}(F \wedge J) \subseteq (\text{free}(H) \triangle \text{free}(J)) - \text{free}(F) \subseteq V$. The other cases are similar.

PROOF (**Theorem 16**) Given $P, Q \in \Sigma_k$ safe,[8] set

$$L(\bar{x}) := (P(\bar{x}) \wedge \exists y\, r(y)) \vee Q(\bar{x})$$

where r does not appear in P or Q and has binding pattern r^{i} and where all relations in P and Q have output-only binding patterns. Then

$$P \sqsubseteq Q \text{ iff } L \text{ is feasible}$$

since if $P \sqsubseteq Q$ then $L \equiv Q$ and, by Lemma 15, Q is feasible. Conversely, if $P \not\sqsubseteq Q$ then there is some tuple $\bar{x}$ for which $P(\bar{x})$ holds, yet $Q(\bar{x})$ doesn't. Whether this tuple is part of the answer set of L depends on r, but we cannot query r since it has an input-only access pattern and we have no binding for y.

Given $P, Q \in \Pi_k$ safe, set

$$L(\bar{x}) := (P(\bar{x}) \wedge \neg\exists y\, r(y)) \vee Q(\bar{x})$$

where r does not appear in P or Q and has binding pattern r^{i} and where all relations in P and Q have output-only binding patterns. The result follows by essentially the same argument as the one used above.

[8]Deciding containment of safe queries is just as hard as deciding containment of general.

PROOF (**Theorem 17**) Assume $Q' \in \Pi_1$ is of the form $\forall \bar{y} Q(\bar{x}\bar{y})$ with Q quantifier-free. By the definition of executable and by Lemma 11, if there is a $\mathcal{V}$-executable query $Q'' \equiv Q'$, then there is such an executable query of the form

$$Q''(\bar{x}) := F(\bar{x}) \wedge \neg \exists \bar{y} G(\bar{x}\bar{y})$$

where F and G are quantifier-free satisfying

- F is $\mathcal{V}$-executable,
- G is $\mathcal{V} \cup \text{vars}(F) \setminus \{\bar{x}\}$-executable,
- $\{\bar{x}\} \notin \text{vars}(F)$, and
- $\text{vars}(G) \subseteq V \cup \text{vars}(F) \cup \{\bar{x}\}$.

By the definition of feasible, we know there are such F and G iff there are F' and G' satisfying the same conditions as above except having "feasible" instead of "executable." Therefore, to determine whether $Q' \in \Pi_1$ is feasible, it is enough to check whether there are such feasible F and G so that

$$Q(\bar{x}\bar{y}) := F(\bar{x}) \wedge \neg \exists \bar{y} G(\bar{x}\bar{y}).$$

The trick is to find F and G efficiently and work with them without explicitly writing them down since otherwise we may incur an exponential increase in the length of the query expression.

If we had $\neg Q$ in DNF so $\neg Q := \bigvee_{1 \leqslant i \leqslant k} Q_i$ then $Q := \bigwedge_{1 \leqslant i \leqslant k} \neg Q_i$ and we could easily extract F and G from Q by looking at the variables in each Q_i:

$$F := \bigwedge_{i \in \mathcal{F}} \neg Q_i \text{ and } G := \bigvee_{i \in \mathcal{G}} Q_i$$

where

$$\mathcal{F} := \{i \mid \bar{x} \notin \text{vars}(Q_i)\} \text{ and } \mathcal{G} := \{1, \ldots, k\} \setminus \mathcal{F}.$$

Clearly we can check the conditions on $\text{vars}(F)$ and $\text{vars}(G)$ efficiently. To check that G is $V \cup \text{vars}(F) \setminus \{\bar{x}\}$-feasible, it is enough to check that, for $i \in \mathcal{G}$, $\neg Q_i$ is $(V \cup \text{vars}(F) \setminus \{\bar{x}\})$-feasible, which we can do in polynomial time. To check that F is $\mathcal{V}$-feasible we need to check that $\text{ans}(F) \sqsubseteq F$. We can do this by checking that, for each conjunction P of literals from Q,

$$(\forall i \in \mathcal{F})(P \sqsubseteq \neg Q_i)$$

implies

$$(\forall i \in \mathcal{F})(\text{ans}(P) \sqsubseteq \neg Q_i)$$

which is a way of verifying

$$P \sqsubseteq F \text{ implies } \text{ans}(P) \sqsubseteq F.$$

Unfortunately, we do not have $\neg Q$ in DNF and transforming $\neg Q$ into DNF may cause an exponential length increase. However, we can efficiently find any Q_i in such DNF form and that is what we use. That is, instead of the above we check that, for all P,

$$\forall B(Q \sqsubseteq B \text{ and } \bar{x} \notin \text{vars}(B) \Longrightarrow P \sqsubseteq \neg B)$$

implies

$$\forall B(Q \sqsubseteq B \text{ and } \bar{x} \notin \text{vars}(B) \Longrightarrow \text{ans}(P) \sqsubseteq \neg B)$$

where P and B are conjunctions of literals from Q (here B plays the role of Q_i). This is the same as checking that, for all P,

$$P \sqsubseteq F \text{ implies } \text{ans}(P) \sqsubseteq F.$$

The details for extracting and testing G are similar (using conjunctions B of literals from Q as above instead of Q_i), but easier since we do not need to work with all of G at once. The required checks for safety are also easy and can be done in **coNP**. We can do all of this in $\mathbf{\Pi}_2^{\text{P}}$ and therefore $\text{FEASIBLE}(\Pi_1, \text{FO}) \in \mathbf{\Pi}_2^{\text{P}}$.