Logic-based Perspectives on Query Reformulation over Restricted Interfaces

Michael Benedikt
University of Oxford, UK
michael.benedikt@cs.ox.ac.uk

ABSTRACT

We overview recent developments on query reformulation over a restricted interface, in the presence of integrity constraints. We overview an approach to the problem via reduction to query containment with constraints, where the reduction makes use of interpolation algorithms from logic. We first present the approach in the context of reformulating one query as another query using a fixed set of tables. We then generalize to reformulation of a query as a plan over a set of access methods.

1. INTRODUCTION

This article summarizes a series of articles [2, 8, 9, 5, 6, 4] revisiting *reformulating queries over restricted data interfaces in the presence of integrity constraints*. In this problem, we start with a *source query* Q written in some declarative language and a set of integrity constraints Σ. We also have some "interface restriction", representing a limit on how data is accessed. We want to translate Q into a target object P — either a query or a program — that satisfies two properties:

- P is equivalent to Q for all query inputs satisfying the constraints Σ
- P satisfies the interface restriction.

We consider two flavors of interface restriction. The first is *vocabulary-based restriction*. We have a subset $\mathcal{V}$ of the tables in the schema, and we want P to be a query referencing only tables in $\mathcal{V}$. The prototypical case is where $\mathcal{V}$ is a set of view tables, and Σ includes the assertion that each view table stores exactly the tuples satisfying the corresponding view definition.

A second kind of interface restriction is given by *access methods*: each table T with n attributes is associated with a set (possibly empty) of access methods. Each method is further associated with a subset of the attributes of T – the input positions. The idea is that a method gives functional access to table T: given a binding for the input positions, it returns the matching tuples in T. Our reformulation problem is to see if Q is equivalent, modulo constraints Σ, to something "executable with respect to the access methods". That is, we want a plan that makes use of the access methods, whose result agrees with Q for all inputs satisfying Σ.

Thus our reformulation problem generalizes both rewriting queries with respect to views, which has been studied for decades, as well as prior work on determining whether a query can be executed with access methods [20, 19, 31, 23, 24, 12].

We present a common framework for dealing with both of these problems, using a reduction to query containment with constraints. From a proof of a query containment one can extract a reformulation, using a technique from logic called *interpolation*. This framework provides new reformulation algorithms both when the target is a query and when the target is a plan. It also gives a common way to see many prior results in the area. For instance, it allows us to re-derive classic results on querying over views, such as Levy, Mendelzon, Sagiv, and Srivastava's [17], and methods for reformulating queries using constraints, such as the Chase and Backchase of Deutsch, Popa, and Tannen [16, 14, 26].

Although the unified presentation of reformulation comes from our own work, it builds on a long line of prior papers. Particularly important is Nash, Segoufin and Vianu's work on characterizing when a query can be expressed using views [25]. Two other key antecedents are Deutsch, Ludäscher, and Nash's paper [12] on querying with access methods and integrity constraints, and the book of Toman and Weddell [30] on reformulation over constraints.

In this survey article we will go through some of the main ideas, skipping most of the details. A full exposition of the techniques, as well as a detailed discussion of related work, can be found in [4].

Organization:. Section 2 contains standard DB preliminaries, as well as some results on query containment with constraints. Section 3 looks at the reformulation problem for vocabulary-based inter-

faces. Section 4 turns to interfaces based on access methods. We present a discussion of implications and future directions in Section 5, before concluding in Section 6.

2. PRELIMINARIES

Data and queries. The basic data model of a querying scenario is given by a *relational schema* $\mathcal{S}$ that consists of a set of *relations* each with an associated *arity* (a positive integer). The *positions* of a relation R of $\mathcal{S}$ are $1, \ldots, n$ where n is the arity of R. An *instance* of R is a set of n-tuples (finite or infinite), and an *instance* I of $\mathcal{S}$ consists of instances for each relation of $\mathcal{S}$. For an instance I of $\mathcal{S}$ and a relation $R \in \mathcal{S}$, the set of tuples assigned to R in I is the *interpretation of R in I*. We can equivalently see I as a set of *facts* $R(a_1 \ldots a_n)$ for each tuple $(a_1 \ldots a_n)$ in the instance of each relation R. The *active domain* of I, denoted $\mathsf{adom}(I)$, is the set of all the values that occur in facts of I.

The source queries that are being reformulated will be *conjunctive queries* (CQs) which are expressions of the form $\exists x_1 \ldots x_k \ (A_1 \wedge \cdots \wedge A_m)$, where the A_i are *relational atoms* of the form $R(x_1 \ldots x_n)$, with R being a relation of arity n and $x_1 \ldots x_n$ being variables or constants. A *union of conjunctive queries* (UCQ) is a disjunction of CQs. The target for reformulation of a CQ will not necessarily be another CQ or even a UCQ. Sometimes it will be a formula of first-order logic (FO). FO is built up from relational atoms and equalities using the boolean operations and quantifiers $\forall$ and $\exists$, where quantifiers always range over the active domain. For an instance I and query Q given by an FO formula, the set of bindings for the variables that satisfy the formula is the *output* of Q on I, denoted $Q(I)$.

Integrity constraints. To express integrity constraints on instances, we will use sentences of FO. Some of the results apply only to *dependencies*, which will be either *tuple-generating dependencies* (TGDs) or *equality-generating dependencies* (EGDs).

A TGD is an FO sentence τ of the form $\forall \vec{x} \ (\varphi(\vec{x}) \rightarrow \exists \vec{y} \ \psi(\vec{x}, \vec{y}))$ where φ and ψ are CQs. An EGD is of the form: $\forall \vec{x} \ (\varphi(\vec{x}) \rightarrow x_i = x_j)$ where φ is a CQ whose variables include x_i and x_j.

For brevity, in the sequel, we will omit outermost universal quantifications in dependencies.

Query containment problems. A *query containment with constraints* is an assertion

$$Q \subseteq_\Sigma Q'$$

where Q and Q' are queries given by logical formulas, and Σ is a set of integrity constraints (given by logical sentences). Such a containment *holds* if for every instance I satisfying Σ, the result of Q on I is contained in the result of Q' on I [1]. We say that "Q is contained in Q' with respect to Σ". To verify a query containment, it is necessary and sufficient to find a *proof* in some suitable proof system. There are many proof systems for first-order logic. One example is the *tableau proof system*. A tableau proof witnesses that a first-order logic formula φ is unsatisfiable. It is a tree where every node p is associated with a set of formulas F_p. The root of the tree is associated with the singleton set of formulas $\{\varphi\}$ and every leaf must be associated to a set containing an explicitly contradictory formula (False). A non-leaf node p, associated with formulas F_p, has at most two children. For each child c, the set of formulas F_c is related to F_p by adding on some subformula of a formula $\gamma_p \in F_p$. For example, if F_p includes a formula γ_p that is a disjunction $\gamma_1 \vee \gamma_2$, then one of the children will contain γ_1 and the other will contain γ_2. Tableau proofs give a complete method for detecting unsatisfiability of a sentence:

Proposition 1: If φ is unsatisfiable there is some tableau proof witnessing this. ■

A query containment $Q_1 \subseteq_\Sigma Q_2$ holds exactly when $Q_1 \wedge \Sigma \wedge \neg Q_2$ is unsatisfiable. Thus tableau proofs provide a complete method to verify query containment with arbitrary first-order integrity constraints Σ. Other proof systems for first-order logic include resolution and natural deduction. For query containment problems in which the constraints Σ consist of *dependencies*, there are more specialized proof systems, such as *the chase* [22].

3. REFORMULATING OVER A SUBSET OF THE RELATIONS

We revisit the reduction from reformulation to query containment with constraints implicit in the work of Nash, Segoufin, and Vianu [25]. We will phrase their work in terms of reformulation of a query defined over a *source* vocabulary, where the goal is to translate it into a query over a *target* vocabulary. We have integrity constraints that can involve relations in both the source vocabulary and the target vocabulary. Thus the "interface" is defined by giving the target vocabulary.

Let $\mathcal{S}$ be a collection of relations, Σ a set of integrity constraints, and $\mathcal{V}$ a subset of $\mathcal{S}$. A *first-order reformulation of Q over $\mathcal{V}$ with respect to Σ*

[1] Note that in this work we will always consider containments over all instances, not just finite ones. The theorems presented here do not hold for general first-order logic if only finite instances are considered. However, for many classes of interest, the results hold verbatim over finite instances [4].

means a safe first-order query $Q_{\mathcal{V}}$ (that is, a query equivalent to a relational algebra query), using only the relations in $\mathcal{V}$, and such that for every instance I satisfying Σ, $Q(I) = Q_{\mathcal{V}}(I)$.

Example 3.1. A university database has a relation Prof containing ids and last names of professors, along with the name of the professor's department. It also has a relation Stud listing the id and last name of each student, along with their advisor's id.

The database does not allow users to access the Prof and Stud relations directly, but instead exposes a view VProf where the id attribute is dropped, and a relation VStud where the advisor's id is replaced with the advisor's last name.

That is, VProf is a view defined by the formula:

$$\{ \mathsf{lname},\ \mathsf{dname} \mid \exists\ \mathsf{profid}\ \mathsf{Prof}(\mathsf{profid},\ \mathsf{lname},\ \mathsf{dname})\}$$

or equivalently by the constraint:

$$(\exists \mathsf{profid}\ \mathsf{Prof}(\mathsf{profid}, \mathsf{lname}, \mathsf{dname})) \leftrightarrow \mathsf{VProf}(\mathsf{lname}, \mathsf{dname})$$

VStud is a view defined by the formula:

$$\{\mathsf{studid}, \mathsf{lname}, \mathsf{profname} \mid \exists \mathsf{profid}\ \exists \mathsf{dname}\ \mathsf{Stud}(\mathsf{studid}, \mathsf{lname}, \mathsf{profid}) \wedge \mathsf{Prof}(\mathsf{profid}, \mathsf{profname}, \mathsf{dname})\}$$

or equivalently by the constraint:

$$[(\exists \mathsf{profid}\ \exists \mathsf{dname}\ \mathsf{Prof}(\mathsf{profid}, \mathsf{profname}, \mathsf{dname}) \wedge \mathsf{Stud}(\mathsf{studid}, \mathsf{lname}, \mathsf{profid})) \leftrightarrow \mathsf{VStud}(\mathsf{studid}, \mathsf{lname}, \mathsf{profname})]$$

Consider the query asking for the names of the advisors of a given student. We can reformulate this query over the VStud view. The reformulation is just the query returning the profname attribute of the view. But a query asking for the last names of all students that have an advisor in the history department can *not* be refomulated using these two views: knowing the advisor's name is not enough to identify the department. ◁

If $\mathcal{L}$ is some subset of FO, we can similarly talk about an $\mathcal{L}$ reformulation of Q over $\mathcal{V}$ with respect to Σ.

One of the main ideas of [25] is a reduction of finding a reformulation to verifying a query containment with constraints, and the key to this reduction is the notion of *determinacy*, which we define next. If Σ is a collection of first-order constraints, we say that a first-order query Q over $\mathcal{S}$ is *determined over* $\mathcal{V}$ *relative to* Σ if:

> For any two instances I and I' that satisfy Σ and have the same interpretation of all relations in $\mathcal{V}$ (that is, they have the same T-facts for each $T \in \mathcal{V}$), we have $Q(I) = Q(I')$.

That is, two instances that agree on $\mathcal{V}$ must agree on Q. How does determinacy connect with reformulation? We first show that determinacy boils down to checking query containment with constraints. Let us extend our original schema for the constraints Σ and the query Q by making a copy R' of every relation R in the schema. Let Q' be the copy of Q on the new relations, and Σ' be the copy of the constraints Σ on the new relations. Our assumption of determinacy of Q can be restated as a query containment problem $Q \subseteq_\Gamma Q'$, where Γ contains Σ, Σ' and the additional *interface axioms*:

$$\bigwedge_{T \in \mathcal{V}} \forall \vec{y}\ T(\vec{y}) \leftrightarrow T'(\vec{y})$$

This is the *query containment corresponding to determinacy of* Q *over* $\mathcal{V}$ *relative to* Σ.

We now want to argue that when the query containment above holds, we can get a reformulation of Q over $\mathcal{V}$ with respect to Σ. To do this, we need to bring into the picture *interpolation*. If we have a query containment problem $Q \subseteq_\Gamma Q'$, and a partition of Γ into Γ_1 and Γ_2, an *interpolant* for the containment and partition is a formula χ such that:

- Q is contained in χ with respect to Γ_1 and χ is contained in Q' with respect to Γ_2
- Every relation in χ occurs in both $\{Q\} \cup \Gamma_1$ and in $\{Q'\} \cup \Gamma_2$.

The crucial facts about interpolants that are relevant to us are:

- *For every query containment problem* $Q \subseteq_\Gamma Q'$*, for every partition of* Γ *into* Γ_1, Γ_2*, there is an interpolant.* Thus in particular if Q is determined over $\mathcal{V}$ relative to Σ, then for every partition of Γ the containment $Q \subseteq_\Gamma Q'$ above corresponding to determinacy has an interpolant.
- Suppose that the query containment corresponding to determinacy of Q over $\mathcal{V}$ relative to Σ holds, and partition Γ above into $\Gamma_1 = \Sigma$, $\Gamma_2 = \Sigma' \cup$ the interface axioms. Let $Q_{\mathcal{V}}$ be any interpolant for the query containment relative to this partition. Then $Q_{\mathcal{V}}$ is a reformulation of Q over $\mathcal{V}$ with respect to Σ.

The first item, saying that interpolants exist, is a basic result in logic; we do not prove it here. We give the short proof of the second item, saying that

interpolants for this particular partition give reformulations. It is a variant of an argument in [11].

PROOF. For simplicity, in the proof we assume Q is a sentence. The definition of interpolant says that $Q_{\mathcal{V}}$ can use only relations that occur both in $\{Q\} \cup \Gamma_1$ and also in $\Gamma_2 \cup \{Q'\}$. This implies that $Q_{\mathcal{V}}$ uses only relations in $\mathcal{V}$.

Since $Q \subseteq_{\Gamma_1} Q_{\mathcal{V}}$, we know that if an instance satisfies the constraints Σ and also satisfies Q, it must also satisfy $Q_{\mathcal{V}}$.

We argue that if an instance satisfies Σ and also satisfies $Q_{\mathcal{V}}$, it must also satisfy Q. Fix I satisfying Σ such that $Q_{\mathcal{V}}$ holds in I. Extend I to an instance $I + I'$ by letting the interpretation of each primed relation R' be the same as the corresponding unprimed relation R of I. The instance $I + I'$ satisfies $Q_{\mathcal{V}}$, Σ', and the interface axioms. Since $Q_{\mathcal{V}} \subseteq_{\Gamma_2} Q$, we know that $I + I'$ satisfies Q'. By the construction of $I + I'$, this means I satisfies Q.

We have shown that for an instance satisfying Σ, Q holds if and only if $Q_{\mathcal{V}}$ holds, which means $Q_{\mathcal{V}}$ is a reformulation.

Since the existence of a reformulation implies that the query containment for determinacy holds, we have the following result:

Theorem 3.1: A conjunctive query Q has a first-order reformulation with respect to vocabulary $\mathcal{V}$ and constraints Σ if and only if Q is determined over $\mathcal{V}$ relative to Σ if and only if the query containment corresponding to determinacy of Q over $\mathcal{V}$ relative to Σ holds. ■

Finding the reformulation. Theorem 3.1 reduces the problem of *existence of a reformulation* to a query containment problem. But of course, we do not just want to know if a reformulation exists, we want to be able to find it. There is a refinement of Theorem 3.1 that talks about finding the reformulation from a witness that the query containment holds. The witness we require is a *proof*. We mentioned in the preliminaries that there are many proof systems for first-order logic, and most of them admit feasible interpolation algorithms. One example is the tableau proof system mentioned in the preliminaries. One can find interpolants quickly from tableau proofs:

Proposition 2: There is a polynomial time algorithm that given a tableau proof that $Q_1 \subseteq_\Sigma Q_2$ and a partition of Σ into Σ_1, Σ_2, finds an interpolant χ for the containment and partition. ■

Using Proposition 2 we can get the refined version of Theorem 3.1 mentioned above:

Theorem 3.2: A conjunctive query Q has a first-order reformulation with respect to vocabulary $\mathcal{V}$ and constraints Σ if and only if the query containment for determinacy of Q over $\mathcal{V}$ holds. Further, given a tableau proof of the query containment we can extract a reformulation in polynomial time. ■

If the constraints Σ are arbitrary first-order sentences, we cannot bound the time taken to find tableau proof, since query containment with first-order constraints is undecidable. But for many restricted classes of constraints – e.g. referential constraints – we can show that the query containment for determinacy is also decidable. Indeed, for many classes C of constraints the query containment problem for determinacy is no more complex than the problem of query containment for the class C.

There is a subtlety we should mention. In our reformulation problems we start with a CQ, and we are interested in reformulations that can be converted to relational algebra. That is, we want reformulations that are not arbitrary first-order, but formulas which only quantify over the active-domain, and where the free variables are safe. Using classical tableau with prior interpolation procedures does not give us this. But by varying both the proof system and the interpolation algorithm slightly, we can get active-domain formulas. Safe reformulations can be achieved by post-processing the interpolants. Details can be found in [4].

3.1 Variation: vocabulary-based reformulation with positive existential queries

We now explore what happens when we restrict the target language for a reformulation. A *positive existential formula with inequalities* ($\exists^{+,\neq}$ formula) is a formula built up using only existential quantification, starting from atomic relations and inequalities. By convention, we also consider the formula False to be positive existential with inequalities. A safe FO formula in this class is equivalent to a relational algebra expression that does not have the difference operator, but allows inequalities in selections. Thus we will sometimes refer to $\exists^{+,\neq}$ formulas as "$USPJ^{\neq}$ queries".

Given a conjunctive query Q, restricted vocabulary $\mathcal{V}$, and constraints Σ given by FO sentences, we are interested in getting a $\exists^{+,\neq}$ reformulation of Q over $\mathcal{V}$ with respect to Σ. This means we want a $\exists^{+,\neq}$ formula over $\mathcal{V}$ that agrees with Q for instances satisfying the constraints.

The query containment for $\exists^{+,\neq}$ reformulation. We start by finding the appropriate variant of determinacy equivalent to a query Q having a $\exists^{+,\neq}$ reformulation. The property was isolated in

[25]. We say that a query Q over schema Sch is *monotonically-determined over* $\mathcal{V}$ relative to Σ if:

> whenever we have two instances I, I' that satisfy Σ and for each relation $T \in \mathcal{V}$, the interpretation of $T \in I$ is a subset of the interpretation of T in I', then $Q(I) \subseteq Q(I')$.

If a $\exists^{+,\neq}$ formula $Q_{\mathcal{V}}$ over $\mathcal{V}$ is true on an instance I, then it is true on any instance I' which only adds tuples to the relations in $\mathcal{V}$. It follows that monotonic-determinacy of Q over $\mathcal{V}$ relative to Σ is a *necessary* condition for Q to have a $\exists^{+,\neq}$ reformulation over $\mathcal{V}$ with respect to Σ.

We can express monotonic-determinacy as a query containment problem. Again we will use a vocabulary that allows us to talk about two copies of the relations, with R' being a copy of R. We let Σ' be a copy of the constraints Σ where each occurrence of a relation R in $\mathcal{S}$ has been replaced by a copy R'. And we let Q' be defined from Q analogously.

Then monotonic-determinacy of a first-order query Q over $\mathcal{V}$ relative to Σ can be restated as saying that the query containment $Q \subseteq_\Gamma Q'$ holds, where Γ contains Σ, Σ' and the "forward interface axiom":

$$\bigwedge_{T \in \mathcal{V}} \forall \vec{y}\ T(\vec{y}) \rightarrow T'(\vec{y}))$$

That is, the difference from determinacy is that we have only implication in the "forward" direction, from unprimed to primed, while for determinacy we have implications in both directions. This is the *query containment for monotonic-determinacy.*

Generating $\exists^{+,\neq}$ reformulations from proofs of the query containment. We now give a result saying that from proofs of the query containment for monotonic-determinacy, we get $\exists^{+,\neq}$ reformulations. It is an analog of Theorem 3.1.

Theorem 3.3: If the constraints Σ are first-order then conjunctive query Q has a $USPJ^{\neq}$ reformulation over $\mathcal{V}$ relative to Σ if and only if the query containment for monotonic-determinacy holds. ■

Theorem 3.3 is proven using the same technique as Theorem 3.1: applying an interpolation algorithm to the query containment associated to monotonic-determinacy. One needs to ensure that the interpolants have some additional properties in order to be sure that the interpolant coming from the proof does not have negation. Many interpolation algorithms are known to ensure this additional property [21]. As with Theorem 3.1, there is a variant that tells us we can find the reformulation effectively given a proof of the query containment.

3.2 Variation: existential reformulation

We now look at another variation of the reformulation problem: determining whether a query can be reformulated using an *existential formula*, or equivalently a UCQ with negation allowed only on atomic formulas. That is, a formula that is built up from atoms *and* negated atoms by positive boolean operators and existential quantification. There are conjunctive queries that are equivalent to existential formulas but not to positive existential ones. For example, in the absence of any constraints $\exists x\ S(x) \wedge \neg R(x)$ is not equivalent to a positive existential formula.

As before, let Sch be a schema with a set of integrity constraints Σ in FO, and $\mathcal{V}$ a subset of the relations of Sch. We start by isolating a determinacy property that Q must have in order to possess an existential reformulation.

For a set of relations $\mathcal{V}$ and instances I and I', we say that I *is a* $\mathcal{V}$ *induced-subinstance of* I' provided for each $T \in \mathcal{V}$ two conditions hold. First, I' contains every fact $T(\vec{c})$ in I. Second, I contains every fact $T(c_1 \ldots c_n)$ in I' such that each c_i occurs in the domain of some relation of $\mathcal{V}$ in I. We say that a query Q over schema Sch is *induced-subinstance-monotonically-determined over* $\mathcal{V}$ relative to Σ if:

> Whenever we have two instances I, I' that satisfy Σ, and I is a $\mathcal{V}$ induced-subinstance of I', then $Q(I) \subseteq Q(I')$.

If an existential formula over $\mathcal{V}$ is true on an instance I, then it is true on any instance I' which only adds tuples to the relations in $\mathcal{V}$ and never "destroys a negated assertion about a relation of $\mathcal{V}$ holding in I". From this we see that if a formula is equivalent to an existential formula under constraints Σ, then the formula is induced-subinstance-monotonically-determined over $\mathcal{V}$ relative to Σ.

As in the previous cases, we can translate this property into a query containment with constraints, but it will be slightly more complicated than determinacy or monotonic-determinacy. Let $\mathsf{InDomain}_{\mathcal{V}}(x)$ abbreviate the formula:

$$\bigvee_{T \in \mathcal{V}} \bigvee_{j} \exists w_1 \ldots \exists w_{j-1}\ \exists w_{j+1} \ldots w_{\mathsf{arity}(T)}\ T(w_1, \ldots w_{j-1}, x, w_{j+1}, \ldots, w_{\mathsf{arity}(T)})$$

So $\mathsf{InDomain}_{\mathcal{V}}$ states that x is in the domain of a relation in $\mathcal{V}$. The *query containment for induced-subinstance-monotonic-determinacy* is $Q \subseteq_\Gamma Q'$, where Γ contains Σ, Σ', and also the following two addi-

tional axioms:

$$\bigwedge_{T\in\mathcal{V}} (\forall \vec{y}\ T(\vec{y}) \rightarrow T'(\vec{y}))$$

$$\bigwedge_{T\in\mathcal{V}} (\forall \vec{y}\ \bigwedge_i \mathsf{InDomain}_{\mathcal{V}}(y_i) \wedge T'(\vec{y}) \rightarrow T(\vec{y})\ )$$

Comparing with the two previous query containments, we have the forward interface axiom, used in the axioms for determinacy and monotone-determinacy. We also have a restriction of the backward interface axiom used in determinacy. It is easy to see that induced-subinstance-monotonic-determinacy is equivalent to this containment holding.

Extracting reformulations from a proof of the query containment. Continuing the prior pattern, we can show that from a proof of the query containment corresponding to induced-subinstance-monotonic-determinacy, we can extract an existential reformulation from it:

Theorem 3.4: If the constraints Σ are in FO, and CQ Q is induced-subinstance-monotonically-determined over $\mathcal{V}$ relative to Σ, then there is an existential formula $\varphi(\vec{x})$ using only relations in $\mathcal{V}$ that is a reformulation of Q with respect to Σ. ■

The bottom line on vocabulary-based reformulation is:

For every target language, we have a different query containment problem. From proofs of the query containment, we can extract reformulations.

When constraints are dependencies, one can use chase proofs to verify the query containment. For chase proofs, extraction of the reformulation from a proof turns out to be very simple (see [9, 4]).

4. ACCESS METHODS

In the previous section the target of reformulation was specified through vocabulary restrictions. We wanted a query that used a fixed set of target relations, perhaps restricted to be positive existential or existential. In this section we deal with a finer notion of reformulation, where the target has to satisfy *access restrictions*.

Access methods are close to the traditional notion of interface in programming languages: a set of functions that access the data. A specification of this interface will be an extended set of metadata describing both the format of the data (e.g. the vocabulary that would be used in queries and constraints) and the access methods (functions that interact with the stored data).

An *access schema* consists of:

- A collection of relations, each of a given arity.
- A finite collection C of schema constants ("Smith", 3, ...). Schema constants represent a fixed set of values that will be known to a user prior to interacting with the data. Values that can be used in queries and constraints should be schema constants, as before. In addition, any fixed values that might be used in plans that implement queries should come from the set of schema constants. For example, a plan that reformulates a query about the mathematics department might involve first putting the string "mathematics" into a directory service.
- For each relation R, a collection (possibly empty) of *access methods*[2]. Each access method mt is associated with a collection (possibly empty) of positions of R – the *input positions* of mt.
- Integrity constraints, which are sentences of first-order logic as before.

Example 4.1. Suppose we have a Profinfo relation containing information about faculty, including their last names, office number, and employee id. We have a restricted interface that requires giving an employee id as an input.

Intuitively, in such a schema we can not find out information about all professors. But if we had a query asking about a particular professor, hardcoding the professor's employee id, we would be able to use the interface to answer it. ◁

An *access* (relative to a schema as above) consists of an access method of the schema and a *method binding* — a function assigning values to every input position of the method. If mt is an access method on relation R with arity n, I is an instance for a schema that includes R, and AccBind is a method binding on mt, then the *output* or *result* of the access (mt, AccBind) on I is the set of n-tuples $\vec{t}$ such that $R(\vec{t})$ holds in I and $\vec{t}$ restricted to the input positions of mt is equal to AccBind.

An access method may be "input-free": have an empty collection of input positions. In this case, the only access that can be performed using the method is with the empty method binding.

The goal is to reformulate source queries in a target language that represents the kind of restricted computation done over an interface given by an access schema. We formalize this as a language of *plans*. Plans are straight-line programs that can perform accesses and manipulate the results of accesses using relational algebra operators. This lan-

[2]Our definition of "access methods" is a variant of the terminology "access patterns" or "binding patterns" found in the database literature.

guage could model, at a high-level, the plans used internally in a database management system. It could also describe the computation done within a data integration system, which might access remote data via a web form or web service and then combine data from different sources using SQL within its own database management system.

Example 4.2. Suppose we have a Profinfo relation with a restricted interface that requires giving an employee id as an input, as in Example 4.1. But we also have a Udirectory relation containing the employee id and last name of every university employee, with an input-free access method. The fact that the directory contains every employee and that a professor is an employee is captured by the integrity constraint stating that every employee id in the Profinfo is also contained in Udirectory.

Suppose we are interested in the query asking for ids of faculty named "Smith":

$$Q = \exists \text{onum } \mathsf{Profinfo}(\mathsf{eid}, \mathsf{onum}, \text{"Smith"})$$

A reformulation is a program using the given methods, where the program is equivalent to Q for all inputs satisfying the integrity constraints Σ.

One can easily see that there is a reformulation of Q using these access methods: we simply access Udirectory to get all the employee ids, then use these to access Profinfo, filtering the resulting tuples to return only those that have name "Smith".

On the other hand, if we did not have access to Udirectory, we can see intuitively that there is no such reformulation. ◁

Formally, we have a plan language with two basic commands. The first is an *access command*. Over a schema Sch with access methods, an access command is of the form:

$$T \Leftarrow_{\mathsf{OutMap}} \mathsf{mt} \Leftarrow_{\mathsf{InMap}} E$$

where:

- E is a relational algebra expression, the *input expression*, over some set of relations not in Sch (henceforward "temporary relations");
- mt is a method from Sch on some relation R;
- InMap, the *input mapping* of the command, is a function from the output attributes of E onto the input positions of mt;
- T, the *output relation* of the command, is a temporary relation;
- OutMap, the *output mapping* of the command, is a bijection from positions of R to attributes of T.

Note that an access command using an input-free method must take the empty relation algebra expression $\emptyset$ as input.

The manipulation of data retrieved by an access is modeled with the other primitive of our plan language, a *middleware query command*. These are of the form $T := Q$, where Q is a relational algebra expression over temporary relations and T is a temporary relation. We use the qualifier "middleware" to emphasize that the queries are performed on temporary relations created by other commands, rather than on relations of the input schema.

A *relational algebra-plan* (or simply, *RA-plan*) consists of a sequence of access and middleware query commands, ending with at most one *return command* of the form Return E, where E is a relational algebra expression.

Example 4.3. We return to Example 4.2 where we had two sources of information. One was Profinfo, which was available through an access method $\mathsf{mt}_{\mathsf{Profinfo}}$ requiring input on the first position. The second was Udirectory, which had an access method $\mathsf{mt}_{\mathsf{Udirectory}}$ requiring no input. Our query Q asked for ids of faculty named "Smith". One plan that is equivalent to Q would be represented as follows

$$\begin{aligned}
&T_1 \Leftarrow \mathsf{mt}_{\mathsf{Udirectory}} \Leftarrow \emptyset \\
&T_2 := \pi_{\mathsf{eid}}(\sigma_{\mathsf{lname}=\text{"Smith"}} T_1) \\
&T_3 \Leftarrow \mathsf{mt}_{\mathsf{Profinfo}} \Leftarrow T_2 \\
&\mathsf{Return}\ \pi_{\mathsf{eid}}(T_3)
\end{aligned}$$

Above we have omitted the mappings in writing access commands, since they can be inferred from the context. ◁

Fragments of the plan language. There are fragments of our plan language, analogs of the standard fragments of relational algebra and first-order logic. In RA-plans, we allowed arbitrary relational algebra expressions in both the inputs to access commands and the middleware query commands. We can similarly talk about $USPJ^{\neq}$*-plans*, where both kinds of commands can only use $USPJ^{\neq}$ queries.

Plans that reformulate queries. We now define what it means for a plan to correctly implement a query. Given an access schema Sch, a plan *reformulates* a query Q with respect to Sch if for every instance I satisfying the constraints of Sch, the output of the plan on I is the same as the output of Q. We often omit the schema from our notation, since it is usually clear from context, saying that a plan PL reformulates Q. Note that this extends the notion of a query $Q_{\mathcal{V}}$ over relations $\mathcal{V}$ reformulating a query Q.

4.1 Reduction to query containment

Recall from Section 3 that a query Q had a reformulation with respect to a vocabulary-based in-

terface $\mathcal{V}$ if and only if the output of Q was determined by the data stored in $\mathcal{V}$. In the case of access methods, we would like to say that Q can be reformulatable using the access methods if and only if it is "determined by the data we can get via the access methods". We will require some auxiliary definitions to formalize what we mean by "the data we can get via the access methods".

Given an instance I for schema Sch the *accessible part of* I, denoted $\mathsf{AccPart}(I)$ consists of all the facts over I that can be obtained by starting with empty relations and iteratively entering values into the access methods. This will be an instance containing a set of facts $\mathsf{Accessed}R(v_1 \dots v_n)$, where R is a relation and $v_1 \dots v_n$ are a subset of the values in the domain of I such that $R(v_1 \dots v_n)$ holds in I. The content of relations $\mathsf{Accessed}R$ will be formed as a limit of inductively-defined sets $\mathsf{Accessed}R_i$. In the inductive process we will also build a set of elements $\mathsf{accessible}_i$. If Sch contains no schema constants, we start the induction with relations $\mathsf{Accessed}R_0$ and $\mathsf{accessible}_0$ empty. We then iterate the following process until a fixpoint is reached:

$$\mathsf{accessible}_{i+1} = \mathsf{accessible}_i \cup \bigcup_{\substack{R \text{ a relation} \\ j \leq \mathsf{arity}(R)}} \pi_j(\mathsf{Accessed}R_i)$$

and

$$\mathsf{Accessed}R_{i+1} = \mathsf{Accessed}R_i \cup \bigcup_{\substack{(R,\{j_1,\dots,j_m\}) \\ \text{there is a method on } R \text{ with inputs } j_1,\dots,j_m}} \{v_1 \dots v_n | R(v_1 \dots v_n) \text{ in } I,\ v_{j_1} \dots v_{j_m} \in \mathsf{accessible}_i\}$$

Above $\pi_j(\mathsf{Accessed}R_i)$ denotes the projection of $\mathsf{Accessed}R_i$ on the j^{th} position. For a finite instance, this induction will reach a fixpoint after $|I|$ iterations, where $|I|$ denotes the number of facts in I. For an arbitrary instance the union of these instances over all i will be a fixpoint.

Assuming Sch does include schema constants, we modify the definition by starting with $\mathsf{accessible}_0$ consisting of the schema constants, rather than being empty.

Above we consider $\mathsf{AccPart}(I)$ as a database instance for the schema with relations accessible and $\mathsf{Accessed}R$. Below we will sometimes refer to the values in the relation accessible as the *accessible values of* I.

Example 4.4. Suppose our schema has a relation Related of arity 2, with an access method $\mathsf{mt}_{\mathsf{Related}}$ with input on the first position of Related. The schema has exactly one schema constant "Jones". Let instance I consist of facts

$$\{\mathsf{Related}(\text{“Jones”}, \text{“Kennedy”}), \mathsf{Related}(\text{“Kennedy”}, \text{“Evans”}), \mathsf{Related}(\text{“Smith”}, \text{“Thompson”})\}$$

We construct the accessible part of I. We begin by computing:

$$\mathsf{AccessedRelated}_0 = \emptyset, \quad \mathsf{accessible}_0 = \{\text{“Jones”}\}$$

That is, initially the accessible part contains no facts and the only accessible constant is the schema constant "Jones".

We can now apply the inductive rules to get after one iteration:

$$\mathsf{AccessedRelated}_1 = \{(\text{“Jones”}, \text{“Kennedy”})\}$$
$$\mathsf{accessible}_1 = \{\text{“Jones”}, \text{“Kennedy”}\}.$$

and after a second iteration:

$$\mathsf{AccessedRelated}_2 = \{(\text{“Jones”}, \text{“Kennedy”}), (\text{“Kennedy”}, \text{“Evans”})\}$$
$$\mathsf{accessible}_2 = \{\text{“Jones”}, \text{“Kennedy”}, \text{“Evans”}\}$$

At this point, we have reached a fixpoint, so the accessible part of I consists of facts

$$\{\mathsf{AccessedRelated}(\text{“Jones”}, \text{“Kennedy”}), \mathsf{AccessedRelated}(\text{“Kennedy”}, \text{“Evans”})\}$$

The accessible values of I are

$$\{\text{“Jones”}, \text{“Kennedy”}, \text{“Evans”}\}$$

◁

Query Q is said to be *access-determined* over Sch if for all instances I and I' satisfying the constraints of Sch with $\mathsf{AccPart}(I) = \mathsf{AccPart}(I')$ we have $Q(I) = Q(I')$. If a query is *not* access-determined, it is obvious that it cannot be reformulated through any plan, since any plan can only read tuples in the accessible part.

Example 4.5. We return to the setting of Example 4.1, where we have a Profinfo relation containing information about faculty, including their last names, office number, and employee id, but with only an access method $\mathsf{mt}_{\mathsf{Profinfo}}$ that requires giving an employee id as an input. We consider again the query Q asking for ids of faculty named "Smith", where "Smith" is a schema constant.

We show that Q is not access-determined. For this, take I to be any instance that contains exactly one tuple, with lastname "Smith", but with an employee id that is not one of the schema constants. Let I' be the empty instance. The accessible parts of I and I' are empty, since in both cases when we enter all the constants we know about in $\mathsf{mt}_{\mathsf{Profinfo}}$,

we get the empty response. But Q has an output on I but no output on I'.

I and I' witness that Q is not access-determined. From this we see that Q can not be reformulated by any plan using only $\mathsf{mt}_{\mathsf{Profinfo}}$. ◁

We now show that access-determinacy reduces to a query containment. Given a schema Sch with constraints Σ and access methods, we form a schema $\mathsf{AcSch}^{\leftrightarrow}(\mathsf{Sch})$ that has only integrity constraints. $\mathsf{AcSch}^{\leftrightarrow}$ will contain two copies of every relation in Sch, with the copy of R denoted as R'. The constraints of $\mathsf{AcSch}^{\leftrightarrow}$ will include all constraints Σ of Sch, a copy Σ' of the constraints on the new relations, and also the following additional axioms, which we call *accessibility axioms*. The first set of axioms, which we call *forward accessibility axioms*, are as follows (universal quantifiers omitted):

$$\bigwedge_{i \leq m} \mathsf{accessible}(x_{j_i}) \wedge R(x_1 \ldots x_n) \rightarrow R'(x_1 \ldots x_n) \wedge \bigwedge_i \mathsf{accessible}(x_i)$$

Above, R is a relation of Sch having an access method with input positions $j_1 \ldots j_m$.

The second set of axioms, *backward accessibility axioms*, just reverses the roles of R and R':

$$\bigwedge_{i \leq m} \mathsf{accessible}(x_{j_i}) \wedge R'(x_1 \ldots x_n) \rightarrow R(x_1 \ldots x_n) \wedge \bigwedge_i \mathsf{accessible}(x_i)$$

where again R is a relation of Sch having an access method with input positions $j_1 \ldots j_m$.

Intuitively, the primed and unprimed copies are a way of writing a statement about two instances. The relation $\mathsf{accessible}$ represents the common accessible values of the two instances. The axioms state that both instances satisfy the constraints, and ensure that their accessible parts are the same. The notation $\mathsf{AcSch}^{\leftrightarrow}(\mathsf{Sch})$ emphasizes that we have constraints from primed to unprimed and vice versa.

As before, we extend the priming notation to queries, letting Q' be obtained from Q by replacing each relation R by R'. The query containment for access-determinacy is then $Q \subseteq_{\mathsf{AcSch}^{\leftrightarrow}(\mathsf{Sch})} Q'$. Analogously to the vocabulary-based case, we can show that this query containment captures the proposed determinacy property, access-determinacy.

We can also show that whenever the query containment for access-determinacy holds, we can extract a plan that reformulates Q:

Theorem 4.1: For any CQ Q and access schema Sch with constraints in FO, the query containment for access-determinacy holds if and only if there is an RA-plan reformulating Q (over instances of Sch). ■

The proof of Theorem 4.1 uses another refinement of interpolation. Theorem 4.1 only talks about discovering whether a plan exists. There is a variation that says we can find the plan given a suitable proof, analogously to the vocabulary-based setting.

4.2 Variation: plans without negation

When we defined the language of RA-plans,we argued that it forms a natural counterpart to relational algebra in the setting where the interface to data is given by a set of access methods. The analog of $\exists^{+,\neq}$ formulas (equivalent to $USPJ^{\neq}$ queries) in the setting of plans are the $USPJ^{\neq}$-plans mentioned earlier, where we do not allow relational algebra difference in any expressions within commands. We will now consider the problem of reformulating a query as a $USPJ^{\neq}$-plan.

We need a variation of determinacy corresponding to a plan that only uses "accessible data" and only uses it monotonically. We say Q is *access-monotonically-determined* over Sch if whenever we have instances I and I' satisfying the constraints of Sch with every fact of $\mathsf{AccPart}(I)$ contained in $\mathsf{AccPart}(I')$, then $Q(I) \subseteq Q(I')$.

The query containment corresponding to access-monotonic-determinacy is simple: we take the same queries Q, Q' as with access-determinacy, but we include only the forward accessibility axioms.

It is easy to verify that the query containment captures access-monotonic-determinacy. And once again, we can take a proof of the query containment and extraction a reformulation, using the appropriate interpolation algorithm:

Theorem 4.2: For any CQ Q and access schema Sch containing constraints specified in FO, there is a $USPJ^{\neq}$-plan reformulating Q (over instances in Sch) if and only if the query containment for access-monotonic-determinacy holds if and only if Q is access-monotonically-determined over Sch. Furthermore, for every tableau proof witnessing the query containment, we can extract a $USPJ^{\neq}$-plan that reformulates Q. ■

4.3 More variations on restrictions based on access methods

Recall that for vocabulary-based restrictions, there was a variation of the technique for existential reformulation: we are looking for a reformulation and

allow it to use negation, but only at the atomic level. There is a similar variation for plans that "only use negation at the atomic level". The definition of such plans is a bit technical, since in the plan language simply restricting query middleware commands to only use atomic negation is not enough. The definitions and the details of the reformulation method can be found in [9, 4].

In our plans we assumed that an access method on relation R returns all the matching tuples on R. Web service access methods may impose *result limits*, setting an upper bound on the number of matching tuples returned. Another variation of the method, described in [1], shows how to find reformulations with access methods that include result bounds.

5. DISCUSSION

We have presented a few theorems that are representative of the reduction to query containment. In this section we go through some of the implications of the results. This will include a discussion of the main theoretical advantage of the technique, the immediate prospects of applying the results in practice, and remarks on the history of the topic.

5.1 Querying over interfaces

Reformulation is a very broad topic, with still many aspects untouched. Dimensions of the problem include:

- *The logical operators allowed in the target of reformulation.* In this article we have looked at three flavors of reformulation depending on the operators allowed. In "first-order" or "relational-algebra" reformulation, negation is allowed in the target. In "monotone" or "positive-existential" reformulation, we want a reformulation that does not use negation. In between is "existential reformulation", in which we allow negation, but not nested. Surely there are many more possibilities for the allowed operators.
- *The notion of interface.* Reformulation is about synthesizing an implementation with a given interface. Here we have dealt with only two, but there are many notions of data interface that can be considered.
- *The class of constraints.* Integrity constraints are implicit in any analysis of reformulation. In the case of reformulation over views, the constraints are just the view definitions. But one can consider much broader or more restricted classes of constraints, and the class considered will impact the algorithms.
- *The reasoning system used to verify that a reformulation exists.* For constraints that are dependencies, the natural reasoning system for proving query containments for reformulation is the chase. But other proof systems can be used even in the case of dependencies, and more general proof systems need to be used once one goes beyond dependencies. We mentioned tableau and resolution as proof systems in some of our results, but there are many proof systems that can be applied.

The majority of prior work has focused on one spot within the space:

- the interface is given by views
- the constraints consist of view definitions and/or weakly-acyclic dependencies
- the target is a monotone query;
- the reasoning system is the chase.

This case is of course important in practice, and it is attractive because it allows intuitive algorithms like the Chase & Backchase (C&B) [16, 14, 26]. The biggest impact of the work presented here is a common framework for exploring a much wider space. This has a conceptual benefit, and provides, at least in principle, reformulation algorithms for classes that have not been considered in the past.

There are many other kinds of data interfaces that could be considered e.g. web interfaces that allow one to send SQL commands; keyword base interfaces. And for other data models there are still further possibilities. The broader framework presented here could be useful for generalizing reformulation to new contexts.

5.2 Practical aspects

The approach based on reduction to query containment has an advantage in its generality. But does it provide better algorithms in practice?

Suppose we specialize this framework to the most well-studied setting: a vocabulary-based interface, the target being $USPJ^{\neq}$ queries ("monotone queries", for short) and constraints that are dependencies where the chase process [22] terminates. If we use the chase as our proof system, and look at the algorithm that results from applying the machinery, what we obtain is a variation of the C&B algorithm mentioned above. The framework by itself only tells us how to find one reformulation. For finding a *good* reformulation, one needs a way of searching through the space of proofs, and then selecting the best one. When constraints are dependencies, there are additional optimizations for efficiently searching the search space, and these have been incorporated into the C&B [16]. In the same way, traditional

algorithms for finding negation-free rewritings over CQ views [18, 27] include important techniques for efficiently enumerating the space of rewritings. One does not get this "for free" from the interpolation framework.

Let us now stick to the vocabulary-based setting, searching for monotone reformulations, but let our constraints be *disjunctive dependencies*, rules with disjunction in the head. One can use an extension of the chase, the "disjunctive chase" [13] as a proof system. Specializing our framework using interpolation to this setting, one gets a variation of the C&B using disjunction [12]. However, there are other proof systems that one can apply, such as tableau proofs or resolution, and applying the framework with these will give different rewritings than those provided by the C&B. Preliminary results [3] show that the interpolation-based approach on top of resolution can give much more succinct reformulations than the approach using the disjunctive chase.

When the constraints go beyond disjunctive dependencies, we know of no competitor to the approach via interpolation. But to make use of the technique here may require more complex theorem proving techniques. Similarly, if we look at finding general first-order reformulations over views, rather than monotone rewritings, we can still apply the technique to reduce to theorem proving, but the theorem proving problem is undecidable in general [15], so we may need to make use of incomplete or non-terminating methods. Further, to find a good reformulation, one needs access to multiple proofs from a theorem prover, and a way to search through these proofs: theorem provers do not have such APIs at present.

One of the simplest practical application of the frameworks is in the case of access methods and integrity constraints in the form of dependencies with terminating chase. In a data integration setting, these constraints may relate local sources that have access methods to an integrated schema. They may also restrict the local sources. Given a query (e.g. over an integrated schema), the access methods, and the constraints, the variant of the approach given in Subsection 4.2 can be applied to determine whether a $USPJ^{\neq}$-plan can be generated, and if so synthesize a plan. We have applied the framework to a number of application settings, ranging from web services [7] to more traditional database access methods [5, 6].

5.3 Algorithms and semantics

Finally, we want to mention a general "lesson learned" from this line of work, concerning the interplay of algorithms and semantics.

There is a long history of algorithmic work for rewriting queries over restricted interfaces. Examples include the work of Levy, Mendelzon, Sagiv, and Srivastava [17], leading to the well known bucket [18] and MiniCon [27] algorithms. The C&B is another example of a clever algorithm for finding reformulations, in the more general setting of queries over a subset of the relations with respect to integrity constraints [16, 14, 26].

A parallel line of research deals with characterizing queries that can be rewritten in certain ways, relating the syntactic restrictions in the target language and semantic properties of the source query. Examples in this line are the homomorphism preservation theorem (see [28]), which states that a first-order formula can be rewritten as a UCQ exactly when it is preserved under homomorphism. In databases, the semantic line includes the work of Segoufin and Vianu [29] and the subsequent TODS paper of Nash, Segoufin, and Vianu [25]. They defined the notion of determinacy we used in Section 3, and showed that it characterizes queries that have relational algebra reformulations.

Another message of this work is that the semantic and algorithmic lines are connected. The semantic approach gives a clean way to see that certain reformulations exists. But it can be converted to an algorithmic technique, applicable not only to view-based reformulation, but to reformulation with integrity constraints and access methods. We think that reformulation gives a nice example of how expressiveness results and algorithmic methods can interact.

6. CONCLUSION

We have presented an overview of a recipe for query reformulation over interfaces. It involves two components: a reduction to query containment problems, and then the use of interpolation algorithms applied to proofs of a containment. We have given an idea of the generality of the framework, showing it is applicable to different kinds of interfaces and different kinds of logical operators in the reformulation target.

A more detailed look at reformulation can be found in Toman and Weddell's book [30], or in the book that takes the perspective presented here, [4]. For the reader interested primarily in the case of TGD constraints, the paper [9] gives a shorter overview.

7. REFERENCES

[1] Antoine Amarilli and Michael Benedikt. When can we answer queries using

result-bounded services. In *PODS*, 2018.
[2] Vince Bárány, Michael Benedikt, and Pierre Bourhis. Access restrictions and integrity constraints revisited. In *ICDT*, 2013.
[3] Michael Benedikt, Egor Kostylev, Fabio Mogavero, and Efthymia Tsamoura. Reformulating queries: theory and practice. In *IJCAI*, 2017.
[4] Michael Benedikt, Julien Leblay, Balder ten Cate, and Efthymia Tsamours. *Generating Proof from Plans: the interpolation-based approach to Query Reformulation*. Morgan Claypool, 2016.
[5] Michael Benedikt, Julien Leblay, and Efi Tsamoura. PDQ: Proof-driven query answering over web-based data. In *VLDB*, 2014.
[6] Michael Benedikt, Julien Leblay, and Efthymia Tsamoura. Querying with access patterns and integrity constraints. *PVLDB*, 8(6):690–701, 2015.
[7] Michael Benedikt, Rodrigo Lopez-Serrano, and Efthymia Tsamoura. Biological web services: Integration, optimization, and reasoning. In *Advances in Bioinformatics and Artificial Intelligence: Bridging the Gap*, 2016.
[8] Michael Benedikt, Balder ten Cate, and Efi Tsamoura. Generating low-cost plans from proofs. In *PODS*, 2014.
[9] Michael Benedikt, Balder ten Cate, and Efi Tsamoura. Generating plans from proofs. In *TODS*, 2016.
[10] E. W. Beth. On Padoa's method in the theory of definitions. *Indagationes Mathematicae*, 15:330 – 339, 1953.
[11] William Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, 22(3):269–285, 1957.
[12] Alin Deutsch, Bertram Ludäscher, and Alan Nash. Rewriting queries using views with access patterns under integrity constraints. *TCS*, 371(3):200–226, 2007.
[13] Alin Deutsch, Alan Nash, and Jeff Remmel. The chase revisited. In *PODS*, 2008.
[14] Alin Deutsch, Lucian Popa, and Val Tannen. Query reformulation with constraints. *SIGMOD Record*, 35(1):65–73, 2006.
[15] Tomasz Gogacz and Jerzy Marcinkowski. The hunt for a red spider: Conjunctive query determinacy is undecidable. In *LICS*, 2015.
[16] Ioana Ileana, Bogdan Cautis, Alin Deutsch, and Yannis Katsis. Complete yet practical search for minimal query reformulations under constraints. In *SIGMOD*, 2014.
[17] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *PODS*, 1995.
[18] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Query-answering algorithms for information agents. In *AAAI*, 1996.
[19] Chen Li. Computing complete answers to queries in the presence of limited access patterns. *VLDB Journal*, 12(3):211–227, 2003.
[20] Chen Li and Edward Chang. Answering queries with useful bindings. *TODS*, 26(3):313–343, 2001.
[21] Roger C. Lyndon. An interpolation theorem in the predicate calculus. *Pacific Journal of Mathemantics*, 9:129–142, 1959.
[22] David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. Testing implications of data dependencies. *TODS*, 4(4):455–469, 1979.
[23] Alan Nash and Bertram Ludäscher. Processing first-order queries under limited access patterns. In *PODS*, 2004.
[24] Alan Nash and Bertram Ludäscher. Processing union of conjunctive queries with negation under limited access patterns. In *EDBT*, 2004.
[25] Alan Nash, Luc Segoufin, and Victor Vianu. Views and queries: Determinacy and rewriting. *TODS*, 35(3), 2010.
[26] Lucian Popa. *Object/Relational Query Optimization with Chase and Backchase*. PhD thesis, U. Penn., 2000.
[27] Rachel Pottinger and Alon Y. Halevy. Minicon: A scalable algorithm for answering queries using views. *VLDB Journal*, 10(2-3):182–198, 2001.
[28] Benjamin Rossman. Homomorphism preservation theorems. *J. ACM*, 55(3), 2008.
[29] Luc Segoufin and Victor Vianu. Views and queries: determinacy and rewriting. In *PODS*, 2005.
[30] David Toman and Grant Weddell. *Fundamentals of Physical Design and Query Compilation*. Morgan Claypool, 2011.
[31] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, V2*. Comp. Sci. Press, 1989.