

When Can We Answer Queries Using Result-Bounded Data Interfaces?

Antoine Amarilli
LTCI, Télécom ParisTech, Université Paris–Saclay

Michael Benedikt
University of Oxford

ABSTRACT

We consider answering queries on data available through *access methods*, that provide lookup access to the tuples matching a given binding. Such interfaces are common on the Web; further, they often have *bounds* on how many results they can return, e.g., because of pagination or rate limits. We thus study *result-bounded methods*, which may return only a limited number of tuples. We study how to decide if a query is *answerable* using result-bounded methods, i.e., how to compute a *plan* that returns all answers to the query using the methods, assuming that the underlying data satisfies some integrity constraints. We first show how to reduce answerability to a query containment problem with constraints. Second, we show “schema simplification” theorems describing when and how result bounded services can be used. Finally, we use these theorems to give decidability and complexity results about answerability for common constraint classes.

1. INTRODUCTION

Web services expose programmatic interfaces to data. Many of these services can be modeled as an *access method*: given a set of arguments for some attributes of a relation, the method returns all matching tuples for the relation.

EXAMPLE 1.1. Consider a Web service that exposes university employee information. The schema has a relation $\text{Prof}(id, name, salary)$ and an access method pr on this relation: the input to pr is the id of a professor, and an access to this method returns the name and salary of the professor. The schema also has a relation $\text{Udirectory}(id, address, phone)$, and an access method ud : it has no input and returns the id , address, and phone number of all university employees.

Our goal is to answer queries using such services. In the setting of Example 1.1, the user queries are posed on the relations Prof and Udirectory , and we wish to answer them using the methods pr and ud . To do so, we can exploit *integrity constraints* that the data is known to satisfy: for instance, the referential constraint τ that says that the id of every tuple in Prof is also in Udirectory .

EXAMPLE 1.2. Consider $Q_1(n) : \exists i \text{ Prof}(i, n, 10000)$, the query that asks for the names of professors with salary 10000. If we assume the integrity constraint τ , we can implement Q_1 as the following plan: first access ud to get the set of all ids , and then access pr with each id to obtain the salary, filtering the results to return only the names with salary 10000. This

plan reformulates Q_1 over the access methods: it is equivalent to Q_1 on all instances satisfying τ , and it only uses pr and ud to access Prof and Udirectory .

Prior work (e.g., [24, 13]) has formalized this reformulation task as an *answerability* problem: given a schema with access methods and integrity constraints, and given a query, determine if we can answer the query using the methods. The query has to be answered in a *complete* way, i.e., without missing any results. This prior work has led to implementations (e.g. [9, 10, 11]) that can determine how to evaluate a conjunctive query using a collection of Web services, by generating a plan that makes calls to the services.

However, all these works assume that whenever we access a Web service, we will always obtain *all* tuples that match the access. This is not a realistic assumption: to avoid wasting resources and bandwidth, virtually all Web services impose a *limit* on how many results they will return. For instance, the ChEBI service (chemical entities of biological interest, see [11]) limits the output of lookup methods to 5000 entries, while IMDb’s web interfaces impose a limit of 10000 [32]. Some services make it possible to request more results beyond the limit, e.g., using pagination or continuation tokens, but there is often a *rate limitation* on how many requests can be made [26, 29, 41], which also limits the total number of obtainable results. Thus, for many Web services, beyond a certain number of results, we cannot assume that all matching tuples are returned. In this work, we introduce *result-bounded* methods to reason on these services.

EXAMPLE 1.3. The ud method in Example 1.1 may be result-bounded, returning at most 100 entries. If this is the case, then the plan of Example 1.2 is not equivalent to Q_1 as it may miss some result tuples.

Result-bounded methods make it very challenging to reformulate queries. Indeed, they are *nondeterministic*: if the number of results is more than the result bound, then the Web service only returns a subset of results, usually according to unknown criteria. For this reason, it is not even clear whether result-bounded methods can be useful to answer queries in a complete way. However, this may be the case:

EXAMPLE 1.4. Consider the schema of Example 1.1 and assume that ud has a result bound of 100 as in Example 1.3. Consider the query $Q_2 : \exists i a p \text{ Udirectory}(i, a, p)$ asking if there is some university employee. We can answer Q_2 with a plan that accesses the ud method and returns true if the output

is non-empty. It is not a problem that ud may omit some result tuples, because we only want to know if it returns something. This gives a first intuition: result-bounded methods are useful to check for the existence of matching tuples.

Further, result-bounded methods can also help under integrity constraints such as keys or functional dependencies:

EXAMPLE 1.5. Consider the schema of Example 1.1 and the access method ud_2 on $Udirectory$ that takes an id as input and returns the address and phone number of tuples with this id . Assume that ud_2 has a result bound of 1, i.e., returns at most one answer when given an id . Further assume the functional dependency ϕ : each employee id has exactly one address, but possibly many phone numbers. Consider the query Q_3 asking for the address of the employee with id 12345. We can answer Q_3 by calling ud_2 with 12345 and projecting onto the address field. Thanks to ϕ , we know that the result will contain the employee’s address, even though only one of the phone numbers will be returned. This gives a second intuition: result-bounded methods are useful when there is a functional dependency that guarantees that some projection of the output is complete.

In this paper, we study how and when we can use result-bounded methods to reformulate queries and obtain complete answers, formalizing in particular the intuition of Examples 1.4 and 1.5. We then show decidability and complexity results for the answerability problem. We focus on two common classes of integrity constraints on databases: *inclusion dependencies* (IDs), as in Example 1.4, and *functional dependencies* (FDs), as in Example 1.5. But we also show results for more expressive constraints: see Table 1 for a summary.

The first step of our study (Section 3) is to reduce the answerability problem to *query containment under constraints*. Such a reduction is well-known in the context of reformulation of queries over views [39], and in answering queries with access methods without result bounds [12]. However, the nondeterminism of result-bounded methods means that we cannot apply these results directly. We nevertheless show that this reduction technique can still be applied in the presence of result bounds. However, the resulting query containment problem involves complex cardinality constraints, so it does not immediately lead to decidability results.

Our second step (Section 4) is to show *schema simplification results*, which explain why some of the result bounds can be ignored for the answerability problem. These results characterize how result-bounded methods are useful: they capture and generalize the examples above. For instance, we show that for constraints given as IDs, then result-bounded methods are only useful as an *existence check* as in Example 1.4. We also show that, for FD constraints, result-bounded methods are only useful to access the *functionally-determined part of the output*, as in Example 1.5. The proofs introduce a technique of *blowing up models*, i.e., we enlarge them to increase the number of outputs of an access, without violating constraints or changing query answers.

Third, in Section 5, we use the simplification results to deduce that answerability is decidable for these constraint classes, and show tight complexity bounds: we show that the problem is NP-complete for FDs, and EXPTIME-complete

for IDs. We refine the latter result to show that answerability is NP-complete for *bounded-width* IDs, which export only a constant number of variables. This refinement is proved using ideas of Johnson and Klug [33], along with a *linearization* technique of potentially independent interest: we show how the constraints used to reason about answerability can be “simulated” with restricted inclusion dependencies.

In Section 6, we study more expressive constraint classes, beyond IDs and FDs. We do so using a weaker form of simplification, called *choice simplification*, which replaces all result bounds by 1: this intuitively implies that the number of results does not matter. We show that it suffices to consider the choice simplification for a huge class of constraints, including all TGDs, and also constraints consisting of FDs and UIDs. In Section 7, we use this technique to show that decidability of monotone answerability holds much more broadly: in particular it holds for a wide range of classes where query containment is decidable. We conclude the paper by giving some limits to schema simplification and decidability of answerability (Section 8), followed by conclusions (Section 9).

Related work. Our paper relates to a line of work about finding plans to answer queries using access methods. The initial line of work considered finding equivalent “executable rewritings” — conjunctive queries where the atoms are ordered in a way compatible with the access patterns. This was studied first without integrity constraints [37, 36], and then with disjunctive TGD constraints [24]. Later [13, 12] formulated the problem of finding a *plan* that answers the query over the access patterns, distinguishing two notions of plans with access methods: one with arbitrary relational operators in middleware and another without the difference operator. They studied the problem of getting plans of both types in the presence of integrity constraints: following [24], they reduced the search for executable rewritings to query containment under constraints. Further, [13, 12] also related the reduction to a semantic notion of determinacy, originating from the work of Nash, Segoufin, and Vianu [39] in the context of views. Our paper extends the reduction to query containment in the presence of result bounds, relying heavily on the techniques of [24, 39, 13, 12].

Non-determinism in query languages has been studied in other contexts [3, 2]. However, the topic of this work, namely, using non-deterministic Web services to implement deterministic queries, has not been studied. Result bounds are reminiscent of *cardinality constraints*, for which the answerability problem has been studied [28]. However, the two are different: whereas cardinality constraints restrict the *underlying data*, result bounds concern the *access methods* to the data, and makes them *non-deterministic*: this has not been studied in the past. In fact, surprisingly, our schema simplification results (in Sections 4 and 6) imply that answerability with result bounds can be decided *without* reasoning about cardinality constraints.

The techniques we use in reducing to a decidable query containment — e.g., the determinacy notions for non-deterministic services and the technique of “blowing up models” — are all specific to the result-bounded setting, and are introduced in this work. The additional technical tools needed for the complexity analysis revolve around analysis of the chase. While many components of this analysis are specific to the con-

straints produced by our problem, the analysis includes a *linearization* method, which we believe may be more generally applicable. The technique relates to the Datalog[±] agenda of getting bounds for query answering with restricted classes of constraints [15, 34, 19]. Our method deals with guarded rules, as in [15]. Linearization can thus be understood as a refinement of a technique from [30]: we isolate classes that can be reduced to well-behaved classes of linear TGDs, where more specialized bounds [33] can be applied.

2. PRELIMINARIES

Data and queries. We consider a *relational signature* \mathcal{S} that consists of a set of *relations* with an associated *arity* (a positive integer). The *positions* of a relation R of \mathcal{S} are $1, \dots, n$ 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} . We equivalently see I as a set of *facts* $R(a_1 \dots a_n)$ for each tuple $(a_1 \dots a_n)$ in the instance of each relation R . A *subinstance* I' of I is an instance that contains a subset of the facts of I . The *active domain* of I , denoted $\text{Adom}(I)$, is the set of all the values that occur in facts of I .

We study *conjunctive queries* (CQs) which are of the form $\exists x_1 \dots \exists x_k (A_1 \wedge \dots \wedge A_m)$, where the A_i are *relational atoms* of the form $R(x_1 \dots x_n)$, with R being a relation of arity n and $x_1 \dots x_n$ being variables or constants. A CQ is *Boolean* if it has no free variables. A Boolean CQ Q *holds* in an instance I exactly when there is a *homomorphism* of Q to I : a mapping h from the variables and constants of Q to $\text{Adom}(I)$ which is the identity on constants and which ensures that, for every atom $R(x_1 \dots x_n)$ in Q , the atom $R(h(x_1) \dots h(x_n))$ is a fact of I . We let $Q(I)$ be the *output* of Q on I , defined in the usual way: if Q is Boolean, this is true if the query holds and false otherwise. A *union of conjunctive queries* (UCQ) is a disjunction of CQs.

Integrity constraints. To express restrictions on instances, we will use fragments of first-order logic (FO), with the active-domain semantics, and where we disallow constants. We will focus on *dependencies*, especially *tuple-generating dependencies* (TGDs) and *functional dependencies* (FDs).

A *tuple-generating dependency* (TGD) is an FO sentence τ of the form: $\forall \vec{x} (\phi(\vec{x}) \rightarrow \exists \vec{y} \psi(\vec{x}, \vec{y}))$ where ϕ and ψ are conjunctions of relational atoms: ϕ is the *body* of τ while ψ is the *head*. For brevity, in the sequel, we will omit outermost universal quantifications in TGDs. The *exported variables* of τ are the variables of \vec{x} which occur in the head. A *full TGD* is one with no existential quantifiers in the head. A *guarded TGD* (GTGD) is a TGD where ϕ is of the form $A(\vec{x}) \wedge \phi'(\vec{x})$ where A is a relational atom containing all free variables of ϕ' . An *inclusion dependency* (ID) is a GTGD where both ϕ and ψ consist of a single atom with no repeated variables. The *width* of an ID is the number of exported variables, and an ID is *unary* (written UID) if it has width 1. For example, $R(x, y) \rightarrow \exists z w S(z, y, w)$ is a UID.

A *functional dependency* (FD) is an FO sentence ϕ written as $\forall \vec{x} \vec{x}' (R(x_1 \dots x_n) \wedge R(x'_1 \dots x'_n) \wedge (\bigwedge_{i \in D} x_i = x'_i) \rightarrow x_j = x'_j)$, with $D \subseteq \{1 \dots n\}$ and $j \in \{1 \dots n\}$. Intuitively, ϕ asserts that position j is *determined* by the positions of D , i.e., when two R -facts match on the positions of D , they must match on

position j as well. We write ϕ as $D \rightarrow j$ for brevity.

Query and access model. We model a collection of Web services as a service schema Sch , which we simply call a *schema*. It consists of (1.) a relational signature \mathcal{S} ; (2.) a set of integrity constraints Σ given as FO sentences; and (3.) a set of *access methods* (or simply *methods*). Each access method mt is associated with a relation R and a subset of positions of R called the *input positions* of mt . The other positions of R are called *output positions* of mt .

In this work, we allow each access method to have an optional *result bound*. If mt has a result bound, then mt is further associated to a positive integer $k \in \mathbb{N}$; we call mt a *result-bounded method*. Informally, the result bound on mt asserts two things: (i) mt returns at most k matching tuples; (ii) if there are no more than k matching tuples, then mt returns all of them. We also allow access methods to have a *result lower bound*, which only imposes point (ii).

An *access* on an instance I consists of a method mt on some relation R and of a *binding* AccBind for I : the binding is a mapping from the input positions of mt to values in $\text{Adom}(I)$. The *matching tuples* M of the access $(\text{mt}, \text{AccBind})$ are the tuples for relation R in I that match AccBind on the input positions of R , and an *output* of the access is a subset $J \subseteq M$. If there is no result bound or result lower bound on mt , then there is only one *valid output* of the access, namely, the one that contains all matching tuples of I : formally $J := M$. If there is a result bound k on mt , then a *valid output* of the access is any subset $J \subseteq M$ such that:

- (i) J has size at most k
- (ii) for any $j \leq k$, if I has $\geq j$ matching tuples, then J has size $\geq j$. Formally, if $|M| \geq j$ then $|J| \geq j$.

If there is a result lower bound of k on mt , then a *valid output* is any subset $J \subseteq M$ satisfying point (ii) above.

We give specific names to two kinds of methods. First, a method is *input-free* if it has no input positions. Second, a method is *Boolean* if all positions are input positions. Note that accessing a Boolean method with a binding AccBind just checks if AccBind is in the relation associated to the method (and result bounds have no effect).

Plans. We use *plans* to describe programs that use the access methods, formalizing them using the terminology of [13, 12]. A *monotone plan* PL is a sequence of *commands* that produce *temporary tables*. There are two types of commands:

- *Query middleware commands*, of the form $T := E$, with T a temporary table and E a monotone relational algebra expression over the temporary tables produced by previous commands. By *monotone*, we mean that E does not use the relational difference operator; equivalently, it is expressed in monotone first-order logic.
- *Access commands*, written $T \Leftarrow_{\text{OutMap}} \text{mt} \Leftarrow_{\text{InMap}} E$, where T is a temporary table, E is a monotone relational algebra expression over previously-produced temporary tables, InMap is an *input mapping* from the output attributes of E to the input positions of mt , mt is a *method* on some relation R , OutMap is an *output mapping* from the positions of R to those of T , and T is an *output table*. We often omit the mappings for brevity.

The *output table* T_0 of PL is indicated by a special command $\text{Return } T_0$ at the end, with T_0 being a temporary table.

We must now define the semantics of PL on an instance I . Because of the non-determinism of result-bounded methods, we will do so relative to an *access selection* for Sch on I , i.e., a function σ mapping each access (mt, AccBind) on I to a set of facts forming a valid output $J := \sigma(\text{mt}, \text{AccBind})$ for the access. The access selection intuitively describes which valid output is chosen when an access to a result-bounded method matches more tuples than the bound. Note that the definition implies that performing the same access twice must return the same result; however, *all our results still hold without this assumption* (see Appendix A for details).

For every access selection σ , we can now define the semantics of each command of PL for σ by considering them in order. For an access command $T \leftarrow_{\text{OutMap}} \text{mt} \leftarrow_{\text{InMap}} E$ in PL, we evaluate E to get a collection C of tuples. For each tuple \bar{t} of C , we use InMap to turn it into a binding AccBind, and we perform the access on mt to obtain $J_{\bar{t}} := \sigma(\text{mt}, \text{AccBind})$. We then take the union $\bigcup_{\bar{t} \in C} J_{\bar{t}}$ of all outputs, rename it according to OutMap, and write it in T . For a middleware query command $T := E$, we evaluate E and write the result in T . The *output* of PL on σ is then the set of tuples that are written to the output table T_0 in the last command of PL.

The *possible outputs* of PL on I are the outputs that can be obtained with some access selection σ . Intuitively, when we evaluate PL, we can obtain any of these outputs, depending on which access selection σ is used.

EXAMPLE 2.1. *The plan mentioned in Example 1.4 would be written:*

$T \leftarrow \text{ud} \leftarrow \emptyset; \quad T_0 := \pi_{\emptyset} T; \quad \text{Return } T_0;$

The first command runs the relational algebra expression $E = \emptyset$ returning the empty set, which gives a trivial binding for ud. The result of accessing ud is stored in a target table T . The second command projects T to the empty set of attributes, and the third command returns the result of this operation. For every instance I , the plan has only one possible output, which describes whether Udirectory is empty in I .

Answerability. Let Sch be a schema consisting of a relational signature, integrity constraints, and access methods, and let Q be a CQ over the relational signature of Sch. A monotone plan PL *answers* Q under Sch if the following holds: for all instances I satisfying the constraints, PL on I has exactly one possible output, which is the query output $Q(I)$. In other words, no matter which access selection σ is used to return tuples, the output of PL evaluated under σ on I is equal to $Q(I)$. Of course, PL can have a single possible output (and answer Q) even if some intermediate command of PL has multiple possible outputs.

We say that Q is *monotone answerable* under schema Sch if there is a monotone plan that answers it. Monotone answerability generalizes notions of reformulation that have been previously studied. In particular, in the absence of constraints and result bounds, it reduces to the notion of a query having an *executable rewriting with respect to access methods*, studied in work on access-restricted querying [37, 36]. In the setting where the limited interfaces simply expose views, monotone answerability corresponds to the well-known notion of *UCQ rewriting* with respect to views [35].

Query containment and chase proofs. We will reduce

answerability to *query containment under constraints*, i.e., checking whether a Boolean CQ Q' follows from another Boolean CQ Q and some constraints Σ . Formally, any instance that satisfies Q and Σ also satisfies Q' . We denote this as $Q \subseteq_{\Sigma} Q'$. There are well-known reductions between query containment with TGDs and the problem of *certain answers* [27, 15] under TGDs. We will not need the definition of certain answers, but we will use some existing upper and lower bounds from this line of work (e.g., from [15, 5]), rephrased to query containment under constraints.

In the case where Σ consists of dependencies, query containment under constraints can be solved by searching for a *chase proof* [27]. Such a proof starts with an instance called the *canonical database of Q* and denoted $\text{CanonDB}(Q)$: it consists of facts for each atom of Q , and its elements are the variables of Q . The proof then proceeds by *firing dependencies*, as we explain next.

A homomorphism τ from the body of a dependency δ into an instance I is called a *trigger* for δ . We say that τ is an *active trigger* if τ cannot be extended to a homomorphism from the head of δ to I . In other words, an active trigger τ witnesses the fact that δ does not hold in I . We can solve this by *firing* the dependency δ on the active trigger τ , which we also call performing a *chase step*, in the following way. If δ is a TGD, the result of the chase step on τ for δ in I is the superinstance I' of I obtained by adding new facts corresponding to an extension of τ to the head of δ , using fresh elements to instantiate the existentially quantified variables of the head: we call these elements *nulls*. If δ is an FD with $x_i = x_j$ in the head, then a chase step yields I' which is the result of identifying $\tau(x_i)$ and $\tau(x_j)$ in I . A *chase sequence* is a sequence of chase steps, and it is a *chase proof* of $Q \subseteq_{\Sigma} Q'$ if it produces an instance where Q' holds.

It can be shown [27] that whenever $Q \subseteq_{\Sigma} Q'$ there is a chase proof that witnesses this. If all chase sequences are finite we say the *chase with Σ on Q terminates*. In this case, we can use the chase to decide containment under constraints.

Variations of answerability. In this work we focus on monotone answerability. An alternative notion is *RA-answerability*, defined using *RA-plans* that allow arbitrary relational algebra expressions in commands. In the body of the paper we focus on monotone answerability, since for database-style constraints and CQ queries we believe it to be the more natural notion. Indeed, CQs are monotone: if facts are added to an instance, the output of a CQ cannot decrease. Thus the bulk of prior work on implementing CQs over restricted limited interfaces, both in theory [35, 24, 37, 36] and in practice [31, 25], has focused on monotone implementations. However, *many of our results extend to answerability with RA-plans* (see Appendix I). Indeed, we can sometimes show that monotone answerability and RA-answerability coincide.

As a second variation, note that we have defined monotone answerability by requiring that the query and plan agree on all instances, finite and infinite. An alternative is to consider equivalence over finite instances only. We say that a plan PL *finitely answers* Q , if for any finite instance I satisfying the integrity constraints of PL, the only possible output of PLs is $Q(I)$; the notion of a query being *finitely monotone answerable* is defined in the obvious way. Both the finite and unrestricted variants have been studied in past work on access

methods [13, 12], just as finite and unrestricted variants of other static analysis problems (e.g., query containment) have long been investigated in database theory (e.g. [33]). The unrestricted variants usually provide a cleaner theory, while the finite variants can be more precise. In this work our goal is to investigate both variants, leaving a discussion of the trade-off between finite and unrestricted answerability for future work. As it turns out, for the database-style dependencies that we consider, the finite variant can be reduced to the unrestricted one. In particular, this reduction holds for constraints Σ that are *finitely controllable*, by which we mean that for every Boolean UCQs Q and Q' , the containment $Q \subseteq_{\Sigma} Q'$ holds if and only if, whenever a finite instance I satisfies Q , then it also satisfies Q' . For such constraints Σ , there is no distinction between the finite and unrestricted versions:

PROPOSITION 2.2. *If Sch is a schema whose constraints are finitely controllable, then any CQ Q that is finitely monotone answerable with respect to Sch is monotone answerable with respect to Sch.*

PROOF. If Q is finitely monotone answerable there is a monotone plan PL that is equivalent to Q over all finite instances. PL can be rewritten as a UCQ. Thus finite controllability implies that PL is equivalent to Q over all instances, and thus Q is monotone answerable. \square

Many of the well-studied classes of dependencies with decidable static analysis problems are finitely controllable. An exception are dependencies consisting of a mix of UIDs and FDs. However, these are known to be finitely controllable once certain dependencies are added, and thus the finite controllability technique can also be applied in this case (see Section 7).

Finally, for simplicity we also look *only at Boolean CQs from here on*. But our results extend straightforwardly to the non-Boolean case.

3. REDUCING TO QUERY CONTAINMENT

We start our study of the monotone answerability problem by reducing it to *query containment under constraints*, defined in the previous section. We explain in this section how this reduction is done. It extends the approach of [24, 13, 12] to result bounds, and follows the connection between answerability and determinacy notions of [39, 12].

The query containment problem corresponding to monotone answerability will capture the idea that *if an instance I_1 satisfies a query Q and another instance I_2 has more “accessible data” than I_1 , then I_2 should satisfy Q as well*. We will first define accessible data via the notion of *accessible part*. We use this to formalize the previous idea as the property of *access monotonic-determinacy*, and show it to be equivalent to monotone answerability. Using access monotonic-determinacy we show that we can simplify the result bounds of arbitrary schemas, and restrict to *result lower bounds* throughout this work. Last, we close the section by showing how to rephrase access monotonic-determinacy with result lower bounds to query containment under constraints.

Accessible parts. We first formalize the notion of “accessible data”. Given a schema Sch with result-bounded methods and an instance I , an *accessible part* of I is any subinstance

obtained by iteratively making accesses until we reach a fixpoint. Formally, we define an accessible part by choosing an access selection σ and inductively defining sets of facts $\text{AccPart}_i(\sigma, I)$ and set of values $\text{accessible}_i(\sigma, I)$ by:

$$\begin{aligned} \text{AccPart}_0(\sigma, I) &:= \emptyset \text{ and } \text{accessible}_0(\sigma, I) := \emptyset \\ \text{AccPart}_{i+1}(\sigma, I) &:= \bigcup_{\substack{\text{mt method,} \\ \text{AccBind binding in } \text{accessible}_i(\sigma, I)}} \sigma(\text{mt}, \text{AccBind}) \end{aligned}$$

$$\text{accessible}_{i+1}(\sigma, I) := \text{Adom}(\text{AccPart}_{i+1}(\sigma, I))$$

Above we abuse notation by considering $\sigma(\text{mt}, \text{AccBind})$ as a set of facts, rather than a set of tuples. These equations define by mutual induction the set of values (accessible) that we can retrieve by iterating accesses and the set of facts (AccPart) that we can retrieve using those values.

The *accessible part* under σ , written $\text{AccPart}(\sigma, I)$, is then defined as $\bigcup_i \text{AccPart}_i(\sigma, I)$. As the equations are monotone, this fixpoint is reached after finitely many iterations if I is finite, or as the union of all finite iterations if I is infinite. When there are no result bounds, there is only one access selection σ , so only one accessible part: it intuitively corresponds to the data that can be accessed using the methods. In the presence of result bounds, there can be many accessible parts, depending on σ .

Access monotonic-determinacy. We now formalize the idea that a query Q is “monotone under accessible parts”. Let Σ be the integrity constraints of Sch. We call Q *access monotonically-determined* in Sch (or AMonDet, for short), if for any two instances I_1, I_2 satisfying Σ , if there is an accessible part of I_1 that is a subset of an accessible part of I_2 , then $Q(I_1) \subseteq Q(I_2)$. Note that when there are no result bounds, this is a unique accessible part of I_1 and of I_2 , and AMonDet says that when the accessible part grows, then Q grows. The definition of AMonDet is justified by the following result:

THEOREM 3.1. *Q is monotone answerable w.r.t. Sch if and only if Q is AMonDet over Sch.*

Without result bounds, this equivalence of monotone answerability and access monotone determinacy is proven in [13, 12], using a variant of Craig’s interpolation theorem. Theorem 3.1 shows that the equivalence extends to schemas with result bounds (see Appendix C.2 for the proof).

In the sequel, it will be more convenient to use an alternative definition of AMonDet, based on the notion of *access-valid* subinstances. A subinstance I_{Accessed} of I_1 is *access-valid* for I_1 on Sch if, for any access (mt, AccBind) performed with a method mt of Sch and with a binding AccBind whose values are in I_{Accessed} , there is a set J of matching tuples in I_{Accessed} such that J is a valid output for the access (mt, AccBind) in I_1 . In other words, for any access performed on I_{Accessed} , we can choose an output in I_{Accessed} which is also valid in I_1 . We can use this notion to rephrase the definition of AMonDet to talk about a common subinstance of I_1 and I_2 that is access-valid:

PROPOSITION 3.2. *For any schema Sch with constraints Σ and result-bounded methods, a CQ Q is AMonDet if and only if for any two instances I_1, I_2 satisfying Σ , if I_1 and I_2 have a common subinstance I_{Accessed} that is access-valid for I_1 , then $Q(I_1) \subseteq Q(I_2)$.*

The proof, given in Appendix C.1, follows from the definitions. The alternative definition of AMonDet is more convenient, because it only deals with a subinstance of I_1 and not with accessible parts. Thus, we will use this characterization of monotone answerability in all the rest of this paper.

Elimination of result upper bounds. The characterization of monotone answerability in terms of AMonDet allows us to prove a key simplification in the analysis of result bounds. Recall that a result bound of k declares both an *upper bound* of k on the number of returned results, and a *lower bound* on them: for all $j \leq k$, if there are j matches, then j must be returned. We can show that the upper bound makes no difference for monotone answerability. Formally, for a schema Sch with integrity constraints and access methods, some of which may be result-bounded, let $\text{ElimUB}(\text{Sch})$ have the same vocabulary, constraints, and access methods as in Sch, but for each access method mt in Sch with result bound of k , mt has instead a *result lower bound* of k in $\text{ElimUB}(\text{Sch})$, i.e., does not impose the upper bound. We can then show:

PROPOSITION 3.3. *Let Sch be a schema with arbitrary constraints and access methods which may be result-bounded. A CQ Q is monotone answerable in Sch if and only if it is monotone answerable in $\text{ElimUB}(\text{Sch})$.*

PROOF. We show the result for AMonDet instead of monotone answerability, thanks to Theorem 3.1 and Proposition 3.2. Consider arbitrary instances I_1 and I_2 that satisfy the constraints, and let us show that any common subinstance I_{Accessed} of I_1 and I_2 is access-valid for I_1 on Sch iff it is access-valid for I_1 on $\text{ElimUB}(\text{Sch})$: this implies the claimed equivalence.

In the forward direction, if I_{Accessed} is access-valid for I_1 on Sch, then clearly it is access-valid for I_1 in $\text{ElimUB}(\text{Sch})$, as any output of an access on I_{Accessed} which is valid for I_1 on Sch is also valid on $\text{ElimUB}(\text{Sch})$.

In the backward direction, assume I_{Accessed} is access-valid for I_1 on $\text{ElimUB}(\text{Sch})$, and consider an access (mt, AccBind) with values of I_{Accessed} . If mt has no result lower bound, then there is only one possible output for the access, and it is valid also for Sch. Likewise, if mt has a result lower bound of k and there are $\leq k$ matching tuples for the access, then the definition of a result lower bound ensures that there is only one possible output, which is again valid for Sch. Last, if there are $> k$ matching tuples for the access, we let J be a set of tuples in I_{Accessed} which is a valid output for the access in (Sch), and take any subset J' of J with k tuples; it is clearly a valid output for the access in Sch. This establishes the backward direction, concluding the proof. \square

Thanks to this, in our study of monotone answerability in the rest of the paper, we only consider result lower bounds.

Reducing to query containment. Now that we have reduced our monotone answerability problem to AMonDet, and eliminated result upper bounds, we explain how to restate AMonDet as a query containment problem. To do so, we will expand the relational signature: we let accessible be a new unary predicate, and for each relation R of the original signature, we introduce two copies R_{Accessed} and R' with the same arity as R . Letting Σ be the integrity constraints in the original signature, we let Σ' be formed by replacing every

relation R with R' . For any CQ Q , we define Q' from Q in the same way. The AMonDet *containment* for Q and Sch is then the CQ containment $Q \subseteq_{\Gamma} Q'$ where the constraints Γ are defined as follows: they include the original constraints Σ , the constraints Σ' on the R' relations, and the following *accessibility axioms* (with an implicit outermost universal quantifier):

- For each method mt that is not result-bounded, letting R be the relation accessed by mt:
 $(\bigwedge_i \text{accessible}(x_i)) \wedge R(\vec{x}, \vec{y}) \rightarrow R_{\text{Accessed}}(\vec{x}, \vec{y})$
where \vec{x} denotes the input positions of mt in R .
- For each method mt with a result lower bound of k , letting R be the relation accessed by mt, for all $j \leq k$:
 $(\bigwedge_i \text{accessible}(x_i)) \wedge \exists^{\geq j} \vec{y} R(\vec{x}, \vec{y}) \rightarrow \exists^{\geq j} \vec{z} R_{\text{Accessed}}(\vec{x}, \vec{z})$
where \vec{x} denotes the input positions of mt in R . Here $\exists^{\geq j} \vec{y} \phi(\vec{x}, \vec{y})$ for a subformula ϕ is a shorthand that means that there exist at least j different values of \vec{y} such that $\phi(\vec{x}, \vec{y})$ holds.
- For every relation R of the original signature:
 $R_{\text{Accessed}}(\vec{w}) \rightarrow R(\vec{w}) \wedge R'(\vec{w}) \wedge \bigwedge_i \text{accessible}(w_i)$

The AMonDet containment simply formalizes the definition of AMonDet. The idea is that R and R' represent the interpretations of the relation symbol R in I_1 and I_2 ; R_{Accessed} represents the interpretation of R in I_{Accessed} ; and accessible represents the active domain of I_{Accessed} . The constraints Γ include Σ and Σ' , which means that I_1 and I_2 both satisfy Σ . The first two accessibility axioms enforce that the selection is access-valid for I_1 : for non-result-bounded methods, accesses to a method mt on a relation R return all the results, while for result-bounded methods it respects the lower bounds. The last accessibility axiom enforces that I_{Accessed} is a common subinstance of I_1 and I_2 and that accessible includes the active domain of I_{Accessed} . Hence, from the definitions, we have:

PROPOSITION 3.4. *Q is monotone answerable with respect to a schema Sch iff the AMonDet containment for Q and Sch holds.*

Note that, for a schema without result bounds, the accessibility axioms above can be rewritten as follows (as in [13, 12]): for each method mt, letting R be the relation accessed by mt and \vec{x} be the input positions of mt in R , we have the axiom:

$$(\bigwedge_i \text{accessible}(x_i)) \wedge R(\vec{x}, \vec{y}) \rightarrow R'(\vec{x}, \vec{y}) \wedge \bigwedge_i \text{accessible}(y_i)$$

EXAMPLE 3.5. *Let us apply the reduction above to the schema of Example 1.1 with the result bound of 100 from Example 1.3. We see that monotone answerability of a CQ Q is equivalent to $Q \subseteq_{\Gamma} Q'$, for Γ containing:*

- the referential constraint from Udirectory into Prof and from Udirectory' into Prof'
- $\text{accessible}(i) \wedge \text{Prof}(i, n, s) \rightarrow \text{Prof}_{\text{Accessed}}(i, n, s)$,
- for all $1 \leq j \leq 100$:
 $\exists \vec{y}_1 \dots \vec{y}_j (\bigwedge_{1 \leq p < q \leq j} \vec{y}_p \neq \vec{y}_q \wedge \text{Udirectory}(\vec{y}_p))$
 $\rightarrow \exists \vec{y}'_1 \dots \vec{y}'_j (\bigwedge_{1 \leq p < q \leq j} \vec{y}'_p \neq \vec{y}'_q \wedge \text{Udirectory}_{\text{Accessed}}(\vec{y}'_p))$
- $\text{Prof}_{\text{Accessed}}(\vec{w}) \rightarrow \text{Prof}(\vec{w}) \wedge \text{Prof}'(\vec{w}) \wedge \bigwedge_i \text{accessible}(w_i)$
and similarly for Udirectory.

Note that the constraint in the third item is quite complex; it contains inequalities and also disjunction, since we write $\vec{y} \neq \vec{z}$ to abbreviate a disjunction $\bigvee_{i \leq |\vec{y}|} y_i \neq z_i$. This makes

it challenging to decide if $Q \subseteq_{\Gamma} Q'$ holds. Hence, our goal in the next section will be to simplify result bounds to avoid such complex constraints.

4. SIMPLIFYING RESULT BOUNDS

The results in Section 3 allow us to reduce the monotone answerability problem to a query containment problem. However, for result bounds greater than 1, the containment problem involves complex cardinality constraints, as illustrated in Example 3.5, and thus we cannot apply standard results or algorithm on query containment under constraints to get decidability “out of the box”. To address this difficulty, we must *simplify* result-bounded schemas, i.e., change or remove the result bounds. We do so in this section, with *simplification* results of the following form: if we can find a plan for a query on a result-bounded schema, then we can find a plan in a *simplification* of the schema, i.e., a schema with simpler result bounds or no result bounds at all.

These simplification results have two benefits. First, they give insight about the use of result bounds, following the examples in the introduction. For instance, our results will show that for most of the common classes of constraints used in databases, the actual numbers in the result bounds never matter at all for answerability. Secondly, they help us to obtain decidability of the monotone answerability problem.

Existence-check simplification. The first way to use result-bounded methods is to check if some tuples exist, as in Example 1.4. We will formalize this as the *existence-check simplification*, where we replace result-bounded methods by Boolean methods that can only do such existence checks.

Given a schema Sch with result-bounded methods, its *existence-check simplification* Sch' is formed as follows:

- The signature of Sch' is that of Sch plus some new relations: for each result-bounded method mt , letting R be the relation accessed by mt , we add a relation R_{mt} whose arity is the number of input positions of mt .
- The integrity constraints of Sch' are those of Sch plus, for each result-bounded method mt of Sch , a new constraint (expressible as two IDs): $R_{mt}(\vec{x}) \leftrightarrow \exists \vec{y} R(\vec{x}, \vec{y})$, where \vec{x} denotes the input positions of mt in R .
- The methods of Sch' are the methods of Sch that have no result bounds, plus one new Boolean method mt' on each new relation R_{mt} , that has no result bounds either.

EXAMPLE 4.1. Recall the schema Sch of Example 1.1 with the result bound of Example 1.3. The existence-check simplification of Sch has a signature with relations $Udirectory$, $Prof$, and a new relation $Udirectory_{ud}$ of arity 1. It has two access methods without result bounds: the method pr on $Prof$ like in Sch , and a Boolean method ud' on $Udirectory_{ud}$. Its constraints are those of Sch , plus the following IDs:

- $Udirectory(i, a, p) \rightarrow Udirectory_{ud}(i)$; and
- $Udirectory_{ud}(i) \rightarrow \exists a p Udirectory(i, a, p)$.

Clearly, every plan that uses the existence-check simplification Sch' of a schema Sch can be converted into a plan using Sch , by replacing the accesses on the Boolean method of R_{mt} to non-deterministic accesses with mt , and only checking whether the result of these accesses is empty. We want to

understand when the converse is true. That is, when a plan on Sch can be converted to a plan on Sch' . For instance, recalling the plan of Example 1.4 that tests whether $Udirectory$ is empty, we can implement it in the existence-check simplification described in Example 4.1. More generally, we want to identify schemas Sch for which *any* CQ having a monotone plan over Sch has a plan on the existence-check simplification Sch' . We say that Sch is *existence-check simplifiable* when this holds: this intuitively means that “result bounded methods of Sch are only useful for existence checks”.

Showing existence-check simplifiability. We first show that existence-check simplifiability holds for schemas like Example 1.2 whose constraints consist of inclusion dependencies:

THEOREM 4.2. *Let Sch be a schema whose constraints are IDs, and let Q be a CQ that is monotone answerable in Sch . Then Q is monotone answerable in the existence-check simplification of Sch .*

This existence-check simplifiability result implies that for schemas with IDs, monotone answerability is decidable even with result bounds. This is because the existence-check simplification of the schema features only IDs and no result bounds, so the query containment problem for AMonDet only features guarded TGDs, which implies decidability. We will show a finer complexity bound in the next section.

To prove Theorem 4.2, we show that if Q is not AMonDet in the existence-check simplification Sch' of Sch , then it cannot be AMonDet in Sch . This suffices to prove the contrapositive of the result, because AMonDet is equivalent to monotone answerability (Theorem 3.1). This claim is shown with a general method of *blowing up models* that we will reuse in all subsequent simplifiability results. We assume that AMonDet does not hold in the simplification Sch' , and consider a *counterexample* to AMonDet for Sch' : two instances I_1, I_2 both satisfying the schema constraints, such that I_1 satisfies Q while I_2 satisfies $\neg Q$, and I_1 and I_2 have a common subinstance $I_{Accessed}$ which is access-valid for I_1 . We enlarge them, by adding additional facts, to a counterexample to AMonDet for the original schema. We formalize this method in the following immediate lemma:

LEMMA 4.3. *Let Sch and Sch' be schemas and Q a CQ that is not AMonDet in Sch' . Suppose that for some counterexample I_1, I_2 to AMonDet for Q in Sch' we can construct instances $I_1^+ \supseteq I_1$ and $I_2^+ \supseteq I_2$ that satisfy the constraints of Sch and have a common subinstance $I_{Accessed}$ that is access-valid for I_1^+ on Sch , and such that I_2^+ has a homomorphism to I_2 . Then Q is not AMonDet in Sch .*

Let us sketch how the blowing-up process of the lemma is used to prove our existence-check simplification result:

PROOF SKETCH FOR THEOREM 4.2. Assume we have a counterexample I_1, I_2 to AMonDet for Q in the simplification Sch' . We will “blow up” I_1 and I_2 to I_1^+ and I_2^+ as explained in Lemma 4.3, ensuring that I_1^+ and I_2^+ have a common subinstance $I_{Accessed}^+$ that is access-valid for I_1^+ in the original schema Sch . For this, we must ensure that each access in $I_{Accessed}^+$ to a result-bounded method returns either no tuples or more tuples than the bound.

Table 1: Summary of results on simplifiability and complexity of monotone answerability

Fragment	Simplification	Complexity
IDs	Existence-check (Theorem 4.2)	EXPTIME-complete (Theorem 5.3)
Bounded-width IDs	Existence-check (see above)	NP-complete (Theorem 5.4)
FDs	FD (Theorem 4.5)	NP-complete (Theorem 5.2)
FDs and UIDs	Choice (Theorem 6.4)	NP-hard (see above) and in EXPTIME (Theorem 7.2)
Equality-free FO	Choice (Theorem 6.3)	Undecidable (Proposition 8.2)
Frontier-guarded TGDs	Choice (see above)	2EXPTIME-complete (Theorem 7.1)

Intuitively, we form I_{Accessed}^+ in two steps. First, we consider all IDs of the form $R_{\text{mt}}(\vec{x}) \rightarrow \exists \vec{y} R(\vec{x}, \vec{y})$ in Sch' , and we chase them “obliviously”; i.e., for every method mt and value for \vec{x} , we create infinitely many facts to instantiate the head, with infinitely many nulls for \vec{y} . We even do this when the trigger is not active, i.e., when witnesses for the head already exist. Let I_{Accessed}^* be the result of this process.

In a second step, we solve the constraint violations that may have been added by creating these new facts. We do so by applying the chase to I_{Accessed}^* in the usual way with all ID constraints of Σ . This yields I_{Accessed}^+ .

We form I_1^+ by unioning I_{Accessed}^+ with I_1 , and similarly form I_2^+ as the union of I_{Accessed}^+ and I_2 . As the constraints are IDs, we can argue that I_1^+ and I_2^+ satisfy Σ , because I_1 , I_2 , and I_{Accessed}^+ do. We can also construct homomorphisms of I_1^+ back to I_1 and I_2^+ back to I_2 , and we can use I_{Accessed}^+ as the common access-valid subinstance. This proves Theorem 4.2. \square

FD simplification. When our constraints include functional dependencies, we can hope for another kind of simplification, generalizing the idea of Example 1.5: an FD can force the output of a result-bounded method to be deterministic on a projection of the output positions. We will define the *FD simplification* to formalize this intuition.

Given a set of constraints Σ , a relation R that occurs in Σ , and a subset P of the positions of R , we write $\text{DetBy}(R, P)$ for the set of positions *determined* by P , i.e., the set of positions i of R such that Σ implies the FD $P \rightarrow i$. In particular, we have $P \subseteq \text{DetBy}(R, P)$. For any access method mt , letting R be the relation that it accesses, we let $\text{DetBy}(\text{mt})$ denote $\text{DetBy}(R, P)$ where P is the set of input positions of mt . Given a schema Sch with result-bounded methods, we can now define its *FD simplification* Sch' as follows:

- The signature of Sch' is that of Sch plus some new relations: for each result-bounded method mt , letting R be the relation accessed by mt , we add a relation R_{mt} whose arity is $|\text{DetBy}(\text{mt})|$.
- The integrity constraints of Sch' are those of Sch plus, for each result-bounded method mt of Sch , a new constraint (expressible as two IDs): $R_{\text{mt}}(\vec{x}, \vec{y}) \leftrightarrow \exists \vec{z} R(\vec{x}, \vec{y}, \vec{z})$, where \vec{x} denotes the input positions of mt and \vec{y} denotes the other positions of $\text{DetBy}(\text{mt})$.
- The methods of Sch' are the methods of Sch that have no result bounds, plus the following: for each result-bounded method mt on relation R in Sch , a method mt' on R_{mt} that has no result bounds and whose input positions are the positions of R_{mt} corresponding to input positions of mt .

Note that the FD simplification is the same as the existence check simplification when the integrity constraints Σ do not imply any FD. Further observe that, even though the methods of Sch' have no result bounds, any access to a new method mt' of Sch' is guaranteed to return at most one result. This is thanks to the FD on the corresponding relation R , and thanks to the constraints that relate R_{mt} and R .

EXAMPLE 4.4. Recall the schema Sch of Example 1.5 and the FD ϕ on Udirectory . The FD simplification of Sch adds a relation $\text{Udirectory}_{\text{ud2}}(\text{id}, \text{address})$, it replaces ud_2 by a method ud_2 on $\text{Udirectory}_{\text{ud2}}$ whose input attribute is id , and it adds the IDs $\text{Udirectory}_{\text{ud2}}(i, a, p) \rightarrow \text{Udirectory}_{\text{ud2}}(i, a)$ and $\text{Udirectory}_{\text{ud2}}(i, a) \rightarrow \exists p \text{Udirectory}(i, a, p)$. The method ud_2 has no result bound, but the IDs above and the FD ϕ ensure that it always returns at most one result.

Since the FD simplification has no result-bounded methods, the query containment problem for the simplification will not use any complex cardinality constraints, in contrast to Example 3.5.

A schema Sch is *FD simplifiable* if every CQ having a monotone plan over Sch has one over the FD simplification of Sch . As for existence-check, if a schema is FD simplifiable, we can decide monotone answerability by reducing to the same problem in a schema without result bounds.

We use a variant of our “blowing-up process” to show that schemas with only FD constraints are FD-simplifiable:

THEOREM 4.5. Let Sch be a schema whose constraints are FDs, and let Q be a CQ that is monotone answerable in Sch . Then Q is monotone answerable in the FD simplification of Sch .

PROOF SKETCH. We start with a counterexample I_1, I_2 to AMonDet for the FD simplification of Sch , i.e., Q holds in I_1 but not in I_2 , and I_1 and I_2 have a common subinstance I_{Accessed} which is access-valid for I_1 . We blow up the accesses on I_1 one after the other, to enlarge I_1 and I_2 to I_1^+ and I_2^+ satisfying the requirements of Lemma 4.3.

We blow up each access by adding tuples to I_1 and I_2 , to ensure that the access has enough common matching tuples in I_1 and I_2 to define a valid output. It suffices to do so for accesses with result-bounded methods mt , and when some matching tuples in I_1 are not in I_2 . In this case, the definition of $\text{DetBy}(\text{mt})$ ensures that matching tuples in I_1 and I_2 agree on positions of $\text{DetBy}(\text{mt})$, but the assumption implies that not all positions are thus determined. Thus, we can add enough tuples to I_1 and I_2 by defining them on $\text{DetBy}(\text{mt})$ like the existing tuples, and putting fresh values in the other

positions. This can be shown to satisfy the FD constraints of Sch. By performing this blow-up process on each access, we obtain I_1^+ and I_2^+ with the access-valid subinstance I_{Accessed}^+ , and we conclude using Lemma 4.3. \square

5. DECIDABILITY OF MONOTONE ANSWERABILITY

Thus far we have seen a general way to reduce monotone answerability problems with result bounds to query containment problems (Section 3). We have also seen schema simplification results for both FDs and IDs, which give us insight into how result-bounded methods can be used (Section 4). We now show that for these two classes of constraints, the reduction to containment and simplification results combine to give decidability results, along with tight complexity bounds. Note that both of these classes are well-known to be finitely controllable; hence, thanks to Proposition 2.2, *all bounds on monotone answerability in this section also apply to finite monotone answerability*.

Decidability for FDs. We first consider schemas whose constraints consist of FDs. We start with an analysis of monotone answerability in the case *without result bounds*:

PROPOSITION 5.1. *We can decide whether a CQ is monotone answerable with respect to a schema without result bounds whose constraints are FDs. The problem is NP-complete.*

PROOF SKETCH. The lower bound already holds without result bounds or constraints [36], so we focus on the upper bound. By Theorem 3.1 and Proposition 3.4, the problem reduces to the AMonDet query containment problem $Q \subseteq_{\Gamma} Q'$ for Sch. As Sch has no result bounds, we can define Γ using the rewriting of the accessibility axioms given after Proposition 3.4. This ensures that Γ only contains FDs and full TGDs from R and accessible to R' and accessible. We can then show that the chase with Γ terminates in polynomially many rounds. Hence, we can decide containment by checking in NP whether Q' holds on the chase result, concluding the proof. \square

We now return to the situation *with result bounds*. We know that schemas with FDs are FD-simplifiable. From this we get a reduction to query containment with no result bounds, but introducing new axioms. We can show that the additional axioms involving R_{mt} and R do not harm chase termination, so that AMonDet is decidable; in fact, it is no harder than CQ evaluation:

THEOREM 5.2. *We can decide whether a CQ is monotone answerable with respect to a schema with result bounds whose constraints are FDs. The problem is NP-complete.*

Decidability for IDs. Second, we consider schemas whose constraints consist of IDs. As we already mentioned, Theorem 4.2 implies decidability for such schemas. We now give the precise complexity bound:

THEOREM 5.3. *We can decide whether a CQ is monotone answerable with respect to a schema with result bounds whose constraints are IDs. Further, the problem is EXPTIME-complete.*

PROOF. Hardness already holds without result bounds [6], so we focus on the upper bound. By Theorem 4.2, we can equivalently replace the schema Sch with its existence-check simplification Sch' , and Sch' does not have result bounds. Further, we can check that Sch' consists only of IDs, namely, those of Sch plus the IDs added in the simplification. The resulting query containment problem only involves guarded TGDs, and thus we can conclude 2EXPTIME complexity from [18]. A finer analysis of the containment problem for ID without result bounds, given in [6], implies an EXPTIME bound, concluding the proof. \square

Complexity for bounded-width IDs. An important practical case for IDs are those whose *width* — the number of exported variables — is bounded by a constant. This includes UIDs. For bounded-width IDs, it was shown by Johnson and Klug [33] that query containment under constraints is NP-complete. A natural question is whether the same holds for monotone answerability. We accordingly conclude the section by showing the following, which is new even in the setting without result bounds:

THEOREM 5.4. *It is NP-complete to decide whether a CQ is monotone answerable with respect to a schema with result bounds whose constraints are bounded-width IDs.*

To show this result, we will again use the fact that IDs are existence-check simplifiable (Theorem 4.2). Using Proposition 3.4 we reduce to a query containment problem with guarded TGDs. But this is not enough to get an NP bound. The problem is that the query containment includes accessibility axioms, which are not IDs. So we cannot hope to conclude directly using [33]. In the rest of this section, we sketch the proof of Theorem 5.4 in the case *without result bounds*, explaining in particular how we handle this problem. See Appendix E.5 for the complete proof of Theorem 5.4.

In the absence of result bounds, the AMonDet query containment problem $Q \subseteq_{\Gamma} Q'$ can be expressed as follows: Γ contains Σ , Σ' , and for each access method mt accessing relation R with input positions \vec{x} there is an accessibility axiom:

$$(\bigwedge_i \text{accessible}(x_i)) \wedge R(\vec{x}, \vec{y}) \rightarrow R'(\vec{x}, \vec{y}) \wedge \bigwedge_i \text{accessible}(y_i)$$

For each method mt , we can rewrite the axiom above by splitting its head, and obtain the following pair of axioms:

- (Truncated Accessibility):
 $(\bigwedge_i \text{accessible}(x_i)) \wedge R(\vec{x}, \vec{y}) \rightarrow \bigwedge_i \text{accessible}(y_i)$
- (Transfer): $(\bigwedge_i \text{accessible}(x_i)) \wedge R(\vec{x}, \vec{y}) \rightarrow R'(\vec{x}, \vec{y})$

We let Δ be the set of the Truncated Accessibility axioms and Transfer axioms that we obtain for all the methods mt .

The constraints of Δ are TGDs but not IDs. However, we will explain how we can take advantage of their structure to *linearize* Δ together with Σ , i.e., construct a set Σ^{Lin} of IDs that “simulate” the chase by Σ and Δ . To define Σ^{Lin} formally, we will change the signature. Let \mathcal{S} be the signature of the relations used in Σ , not including the special unary relation accessible used in Δ ; and let $w \in \mathbb{N}$ be the constant bound on the width of the IDs in Σ . We expand \mathcal{S} to the signature \mathcal{S}^{Lin} as follows. For each relation R of arity n in \mathcal{S} , we consider each subset P of the positions of R of size at most w , and we also consider the subset $P = \{1 \dots n\}$ even if $n > w$. For each such subset P , we add a relation R_P of arity n to \mathcal{S}^{Lin} .

Intuitively, an R_P -fact denotes an R -fact where the elements in the positions of P are those for which accessible holds. Formally, to translate from \mathcal{S}^{Lin} back to \mathcal{S} , given a set of facts I over \mathcal{S}^{Lin} , we define $\text{UnLin}(I)$ as the set of facts over $\mathcal{S} \cup \{\text{accessible}\}$ obtained by taking each $R_P(\vec{c})$ of I and replacing it by the facts $\{R(\vec{c})\} \cup \bigcup_{i \in P} \{\text{accessible}(c_i)\}$.

Remember that our goal is to linearize Σ and Δ to a set of IDs Σ^{Lin} which emulates the chase by Σ and Δ , up to applying UnLin . If we could ensure that Σ^{Lin} has bounded width, we could then conclude using the result of [33]. We will not be able to enforce this, but Σ^{Lin} will instead satisfy a notion of *bounded semi-width* that we now define. The *basic position graph* of Σ^{Lin} is the directed graph whose nodes are the positions of relations in Σ^{Lin} with an edge from position i of a relation T to position j of a relation U if and only if the following is true: there is an ID $\delta \in \Sigma$ whose body atom A uses relation T , whose head atom A' uses relation U , and there is an exported variable x that occurs at position i of A and at position j of A' . We say that Σ^{Lin} has *semi-width* bounded by w if it can be decomposed into $\Sigma_1^{\text{Lin}} \cup \Sigma_2^{\text{Lin}}$ where Σ_1^{Lin} has width bounded by w and the position graph of Σ_2^{Lin} is acyclic.

We can now state our linearization result:

PROPOSITION 5.5. *Given the set Σ of IDs of width w and the set Δ of Truncated Accessibility and Transfer axioms, and given a set of facts I_0 , we can compute in PTIME a set of IDs Σ^{Lin} of semi-width w and a set of facts I_0^{Lin} satisfying the following: for any set of facts I derivable from I_0 by chasing with Σ and Δ , there is a set of facts I' that can be derived from I_0^{Lin} by chasing with Σ^{Lin} such that $\text{UnLin}(I')$ is a homomorphic image of I .*

PROOF SKETCH. We can translate each of the truncated axioms into IDs on the expanded signature in a straightforward way, but we also need to account for the propagation of accessible-facts via IDs in the chase. We do this by incorporating to Σ^{Lin} some new IDs in the extended signature \mathcal{S}^{Lin} that are implied by Σ and Δ . A saturation algorithm can compute them in polynomial time, thanks to the polynomial bound on the number of subsets P considered in \mathcal{S}^{Lin} . \square

The bound on the semi-width of Σ^{Lin} then implies an NP bound on query containment, thanks to the following easy generalization of the result of Johnson and Klug [33]:

PROPOSITION 5.6. *For fixed w , there is an NP algorithm for containment under IDs of semi-width at most w .*

This allows us to conclude the proof of Theorem 5.4:

PROOF SKETCH OF THEOREM 5.4. NP-hardness already holds without constraints or result bounds [36], so we focus on NP-membership. In the case without result bounds, we have explained how to reduce to a query containment problem $Q \subseteq_{\Gamma} Q'$ with $\Gamma = \Sigma \cup \Sigma' \cup \Delta$. Now, we have shown in Proposition 5.5 how Σ and Δ can be simulated by a set Σ^{Lin} of IDs of bounded semi-width. Further, Σ' consists of bounded-width IDs, so we can modify Σ^{Lin} to incorporate Σ' . This allows us to decide the problem $Q \subseteq_{\Gamma} Q'$ in NP using Proposition 5.6. The details of this argument, and its extension to the case with result bounds, are in Appendix E.5. \square

6. SCHEMA SIMPLIFICATION FOR EXPRESSIVE CONSTRAINTS

We have presented in Section 4 the two kinds of simplifications anticipated in the introduction: *existence-check simplification* (using result-bounded methods to check for the existence of tuples, as in Example 1.4); and *FD simplification* (using them to retrieve functionally determined information, as in in Example 1.5). A natural question is then to understand whether these simplifications capture *all* the ways in which result-bounded methods can be useful, for integrity constraints expressed in more general constraint languages. It turns out that this is not the case when we move even slightly beyond IDs:

EXAMPLE 6.1. *Consider a schema Sch with TGD constraints $T(y) \wedge S(x) \rightarrow T(x)$ and $T(y) \rightarrow \exists x S(x)$. We have an input-free access method mt_S on S with result bound 1 and a Boolean access method mt_T on T . Consider the query $Q = \exists y T(y)$. The following monotone plan answers Q :*

$T_1 \Leftarrow \text{mt}_S \Leftarrow \emptyset; T_2 \Leftarrow \text{mt}_T \Leftarrow T_1; T_3 := \pi_{\emptyset} T_2; \text{Return } T_3;$

That is, we access S and return true if the result is in T .

On the other hand, consider the existence-check simplification Sch' of Sch . It has an existence-check method on S , but we can only test if S is non-empty, giving no indication whether Q holds. So Q is not answerable in Sch' . The same holds for the FD simplification Sch'' of Sch , because Sch implies no FDs, so Sch' and Sch'' are the same.

Thus, existence-check simplification and FD simplification no longer suffice for more expressive constraints. In this section, we introduce a new notion of simplification, called *choice simplification*. We will show that it allows us to simplify schemas with very general constraint classes, in particular TGDs as in Example 6.1. In the next section, we will combine this simplification with our query containment reduction (Proposition 3.4) to show decidability of monotone answerability for much more expressive constraints. Intuitively, choice simplification changes the *value* of all result bounds, replacing them by one; this means that the number of tuples returned by result-bounded methods is not important, provided that we obtain at least one if some exist. We formalize the definition in this section, and show choice simplifiability for two constraint classes: equality-free first-order logics (which includes in particular TGDs), and UIDs and FDs. We study the decidability and complexity consequences of these results in the next section.

Choice simplification. Given a schema Sch with result-bounded methods, its *choice simplification* Sch' is defined by keeping the relations and constraints of Sch , but changing every result-bounded method to have bound 1. That is, every result-bounded method of Sch' returns \emptyset if there are no matching tuples for the access, and otherwise selects and returns one matching tuple. We call Sch *choice simplifiable* if any CQ having a monotone plan over Sch has one over Sch' . This implies that the value of the result bounds never matters.

Choice simplifiability is weaker than existence check or FD simplifiability, but it still has a dramatic impact on the resulting query containment problem:

EXAMPLE 6.2. Consider the schema Sch in Example 1.1 and its naïve axiomatization in Example 3.5. As Sch is choice simplifiable, we can axiomatize its choice simplification instead, and the problematic axiom in the third bullet item becomes a simple ID: $Udirectory(\vec{y}) \rightarrow \exists \vec{y}' Udirectory_{Accessed}(\vec{y}')$.

Showing choice simplifiability. We now give a result showing that choice simplification holds for a huge class of constraints: all first-order constraints that do not involve equality. This result implies, for instance, that choice simplification holds for integrity constraints expressed as TGDs:

THEOREM 6.3. Let Sch be a schema whose constraints are in equality-free first-order logic (e.g., TGDs), and let Q be a CQ that is monotone answerable in Sch . Then Q is monotone answerable in the choice simplification of Sch .

PROOF SKETCH. We use a simpler variant of the “blow-up” method of Theorem 4.2. We start with counterexample models to AMonDet in the choice simplification, and blow them up by cloning the output tuples of each result-bounded access, including all facts that hold about these output tuples. \square

Choice simplifiability with UIDs and FDs. The previous result does not cover FDs. However, we can also show a choice simplifiability result for FDs and UIDs:

THEOREM 6.4. Let Sch be a schema whose constraints are UIDs and arbitrary FDs, and Q be a CQ that is monotone answerable in Sch . Then Q is monotone answerable in the choice simplification of Sch .

PROOF SKETCH. We use a strengthening of the enlargement process of Lemma 4.3 which constructs I_1^+ and I_2^+ from I_1 and I_2 in successive steps, to fix accesses one after the other. The construction that performs the blow-up is more complex (see Appendix F.2): it involves copying access outputs and chasing with UIDs in such a way as to avoid violating the FDs. \square

7. DECIDABILITY USING CHOICE SIMPLIFICATION

In this section, we present the consequences of the choice simplifiability results of the previous section, in terms of decidability for expressive constraint languages. Again, these will apply to both monotone answerability and finite monotone answerability.

Decidable equality-free constraints. Theorem 6.3 implies that monotone answerability is decidable for a wide variety of schemas. The approach applies to constraints that do not involve equality and have decidable query containment. We state here one complexity result for the class of *frontier-guarded TGDs*. These are TGDs whose body contains a single atom including all exported variables. But the same approach applies to *weakly-acyclic TGDs* [27], as well as extensions of FGTGDs with disjunction and negation [14, 7].

THEOREM 7.1. We can decide whether a CQ is monotone answerable with respect to a schema with result bounds whose constraints are frontier-guarded TGDs. The problem is 2EXPTIME-complete.

PROOF. Hardness already holds because of a reduction from query containment under frontier-guarded TGDs (see, e.g., Prop. 3.16 in [12]), already in the absence of result bounds, so we focus on 2EXPTIME-membership. By Theorem 6.3 we can assume that all result bounds are one, and by Proposition 3.3 we can replace the schema with the relaxed version containing only result lower bounds. Now, a result lower bound of 1 can be expressed as an ID. Thus, Proposition 3.4 allows us to reduce monotone answerability to a query containment problem with additional frontier-guarded TGDs, and this is decidable in 2EXPTIME (see, e.g., [8]). \square

Complexity with UIDs and FDs. We now turn to constraints that consist of UIDs and FDs, and use the choice simplifiability result of Theorem 6.4 to derive complexity results for monotone answerability with result-bounded access methods:

THEOREM 7.2. We can decide monotone answerability with respect to a schema with result bounds whose constraints are UIDs and FDs. The problem is in EXPTIME.

Compared to Theorem 5.4, this result restricts to UIDs rather than IDs, and has a higher complexity, but it allows FD constraints. To the best of our knowledge, this result is new even in the setting without result bounds.

PROOF SKETCH OF THEOREM 7.2. We prove only decidability in 2EXPTIME. The finer bound is in Appendix G.2, and uses a more involved variant of our linearization method.

We use choice simplifiability (Theorem 6.4) to assume that all result bounds are one, use Proposition 3.3 to replace them by result lower bounds, and use Proposition 3.4 to reduce to a query containment problem $Q \subseteq_{\Gamma} Q'$. The constraints Γ include Σ , its copy Σ' , and accessibility axioms:

- $(\bigwedge_i \text{accessible}(x_i)) \wedge R(\vec{x}, \vec{y}) \rightarrow R_{\text{Accessed}}(\vec{x}, \vec{y})$ for each non-result-bounded method mt accessing relation R and having input positions \vec{x} ;
- $(\bigwedge_i \text{accessible}(x_i)) \wedge \exists \vec{y} R(\vec{x}, \vec{y}) \rightarrow \exists \vec{z} R_{\text{Accessed}}(\vec{x}, \vec{z})$ for each result-bounded method mt accessing relation R and having input positions \vec{x} ;
- $R_{\text{Accessed}}(\vec{w}) \rightarrow R(\vec{w}) \wedge R'(\vec{w}) \wedge \bigwedge_i \text{accessible}(w_i)$ for each relation R .

Note that Γ includes FDs and non-unary IDs; containment for these in general is undecidable [38]. To show decidability, we will explain how to rewrite these axioms in a way that makes Γ *separable* [21]. That is, we will be able to drop the FDs of Σ and Σ' without impacting containment. First, by inlining R_{Accessed} , we can rewrite the axioms as follows:

- for each non-result-bounded method mt accessing relation R with input positions \vec{x} , $(\bigwedge_i \text{accessible}(x_i)) \wedge R(\vec{x}, \vec{y}) \rightarrow R'(\vec{x}, \vec{y}) \wedge \bigwedge_i \text{accessible}(y_i)$
- for each result-bounded method mt accessing relation R with input positions \vec{x} , $(\bigwedge_i \text{accessible}(x_i)) \wedge R(\vec{x}, \vec{y}) \rightarrow \exists \vec{z} R(\vec{x}, \vec{z}) \wedge R'(\vec{x}, \vec{z}) \wedge \bigwedge_i \text{accessible}(z_i)$

We then modify the second type of axiom so that, in addition to the variables \vec{x} at input positions of mt in R , they also export the variables at positions of R that are determined by the input positions. This rewriting does not impact the soundness of the chase, because each chase step with a rewritten axiom can be mimicked by a step with an original axiom followed by FD applications.

After this rewriting, a simple induction on proof length (see Appendix G.2) shows that firing TGD triggers in the chase never creates a violation on R or R' of the FDs of Σ and Σ' . Hence, after having applied these FDs to Q , we know that we can drop them without impacting query containment. Let Q^* be the minimization of Q under the FDs, and letting Γ^{Sep} denote the rewritten constraints without the FDs. We have shown that monotone answerability is equivalent to $Q^* \subseteq_{\Gamma^{\text{Sep}}} Q'$. As Γ^{Sep} contains only GTGDs, we can infer decidability in EXPTIME using [15], concluding the proof. \square

Extending to finite monotone answerability. extend to monotone answerability over finite instances. For Theorem 7.1 this follows from Proposition 2.2. For Theorem 7.2, we can also show that monotone answerability is decidable in the finite:

COROLLARY 7.3. *We can decide whether a CQ is finitely monotone answerable with respect to a schema with result bounds whose constraints are UIDs and FDs. The problem is in EXPTIME.*

However, constraints that mix UIDs and FDs are not finitely controllable, so we cannot simply use Proposition 2.2. Instead, we will consider the *finite closure* of the set of UIDs and FDs Σ . This is the set Σ^* of FDs and UIDs that are implied by Σ over finite instances. The finite closure of Σ is computable (see [23]), and query containment over finite instances with Σ is equivalent to query containment over all instances with Σ^* :

THEOREM 7.4 ([4]). *For any Boolean UCQs Q and Q' , the following are equivalent: (i.) for any finite instance I satisfying Σ , if Q holds on I then Q' holds on I ; (ii.) for any instance I satisfying Σ^* , if Q holds on I then Q' holds on I .*

This allows us to prove Corollary 7.3:

PROOF SKETCH OF COROLLARY 7.3. We will argue only for decidability. See Appendix G.3 for details and for the EXPTIME bound. Let Sch be a schema whose constraints Σ are UIDs and FDs, and let Sch^* be the same schema as Sch but with constraints Σ^* . We will show that any CQ Q is finitely monotone answerable over Sch iff Q is monotone answerable over Sch^* . We can decide the latter by Theorem 7.2, so it suffices to show the equivalence. For the forward direction, given a monotone plan PL that answers Q over finite instances satisfying Σ , we can convert it to a UCQ Q_{PL} . Now, since Q and Q_{PL} are equivalent over finite instances satisfying Σ , they are equivalent over all instances satisfying Σ^* , thanks to Theorem 7.4. Thus Q is monotone answerable over Σ^* . Conversely, if Q is monotone answerable over Σ^* , it is finitely monotone answerable over all finite instances satisfying Σ^* , but Theorem 7.4 says that finite instances that satisfy Σ must also satisfy Σ^* , which concludes. \square

8. GENERAL FO CONSTRAINTS

We have shown that, for many expressive constraint classes, the value of result bounds does not matter, and monotone answerability is decidable. A natural question is then to understand what happens with schema simplification and decidability for general FO constraints. In this case, we find that choice simplifiability no longer holds:

EXAMPLE 8.1. *Consider a schema Sch with two relations P and U of arity 1. There is an input-free method mt_P on P with result bound 5, and an input-free method mt_U on U with no result bound. The first-order constraints Σ say that P has exactly 7 tuples, and if one of the tuples is in U , then 4 of these tuples must be in U . Consider the query $Q : \exists x P(x) \wedge U(x)$. The query is monotone answerable on Sch : the plan simply accesses P with mt_P and intersects the result with U using mt_U . Thanks to Σ , this will always return the correct result.*

In the choice simplification Sch' of Sch , all we can do is access mt_U , returning all of U , and access mt_P , returning a single tuple. If this tuple is not in U , we have no information on whether or not Q holds. Hence, we can easily see that Q is not answerable on Sch' .

The constraints in the previous example still lie in a decidable language, namely, two-variable logic with counting quantifiers [40]. We can still decide monotone answerability for this language even without any schema simplification; see Appendix H.1. Unsurprisingly, if we move to constraints where containment is undecidable, then the monotone answerability problem is also undecidable, even in cases such as equality-free FO which are choice simplifiable:

PROPOSITION 8.2. *It is undecidable to check if Q is monotone answerable with respect to equality-free FO constraints.*

The same holds for other constraint languages where query containment is undecidable, such as general TGDs.

9. SUMMARY AND CONCLUSION

We formalized the problem of answering queries in a complete way by accessing Web services that only return a bounded number of answers to each access, assuming integrity constraints on the data. We showed how to reduce this to a standard reasoning problem, query containment with constraints. We have further shown simplification results for many classes of constraints, limiting the ways in which a query can be answered using result-bounded plans, thus simplifying the corresponding query containment problem. By coupling these results with an analysis of query containment, we have derived complexity bounds for monotone answerability under several classes of constraints. Table 1 summarizes which simplifiability result holds for each constraint class, as well as the decidability and complexity results.

We have restricted to *monotone* plans throughout the paper. As explained in Appendix I, the reduction to query containment still applies to plans that can use negation. Our schema simplification results also extend easily to answerability with such plans, but lead to a more involved query containment problem. Hence, we do not know how to show decidability of the answerability problem for UIDs and FDs with such plans. We also leave open the question of whether choice simplifiability holds for general FDs and IDs (not UIDs).

In our study of the answerability problem, we have also introduced technical tools which could be useful in a wider context. One example is the blowing-up method that we use in schema simplification results; a second example is linearization, for which we intend to study further applications.

10. REFERENCES

- [1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*. Addison-Wesley, 1995.
- [2] S. Abiteboul, E. Simon, and V. Vianu. Non-deterministic languages to express deterministic transformations. In *PODS*, 1990.
- [3] S. Abiteboul and V. Vianu. Non-Determinism in Logic-Based Languages. *Ann. Math. Artif. Intell.*, 3(2-4), 1991.
- [4] A. Amarilli and M. Benedikt. Finite open-world query answering with number restrictions. In *LICS*, 2015.
- [5] J. Baget, M. Leclère, and M. Mugnier. Walking the decidability line for rules with existential variables. In *KR*, 2010.
- [6] V. Bárány, M. Benedikt, and P. Bourhis. Access patterns and integrity constraints revisited. In *ICDT*, 2013.
- [7] V. Bárány, B. t. Cate, and L. Segoufin. Guarded negation. *Journal of the ACM*, 62(3), 2015.
- [8] V. Bárány, G. Gottlob, and M. Otto. Querying the guarded fragment. In *LICS*, 2010.
- [9] M. Benedikt, J. Leblay, and E. Tsamoura. PDQ: Proof-driven query answering over Web-based data. In *VLDB*, 2014.
- [10] M. Benedikt, J. Leblay, and E. Tsamoura. Querying with access patterns and integrity constraints. In *VLDB*, 2015.
- [11] M. Benedikt, R. Lopez-Serrano, and E. Tsamoura. Biological Web services: Integration, optimization, and reasoning. In *Workshop on Advances in Bioinformatics and Artificial Intelligence: Bridging the Gap*, 2016.
- [12] M. Benedikt, B. ten Cate, J. Leblay, and E. Tsamoura. *Generating plans from proofs: the interpolation-based approach to query reformulation*. Morgan Claypool, 2016.
- [13] M. Benedikt, B. ten Cate, and E. Tsamoura. Generating plans from proofs. In *TODS*, 2016.
- [14] P. Bourhis, M. Manna, M. Morak, and A. Pieris. Guarded-based disjunctive tuple-generating dependencies. *ACM Trans. Database Syst.*, 41(4):27:1–27:45, 2016.
- [15] A. Cali, G. Gottlob, and M. Kifer. Taming the infinite chase: Query answering under expressive relational constraints. In *KR*, 2008.
- [16] A. Cali, G. Gottlob, and M. Kifer. Taming the infinite chase: Query answering under expressive relational constraints. *Journal of Artificial Intelligence Research*, 2013.
- [17] A. Cali, G. Gottlob, and T. Lukasiewicz. A general Datalog-based framework for tractable query answering over ontologies. *Journal of Web Semantics*, 14, 2012.
- [18] A. Cali, G. Gottlob, T. Lukasiewicz, and A. Pieris. A logical toolbox for ontological reasoning. *SIGMOD Record*, 40(3), 2011.
- [19] A. Cali, G. Gottlob, and A. Pieris. Towards more expressive ontology languages: The query answering problem. *Artif. Intell.*, 193, 2012.
- [20] A. Cali, D. Lembo, and R. Rosati. Query rewriting and answering under constraints in data integration systems. In *IJCAI*, 2003.
- [21] A. Cali, D. Lembo, and R. Rosati. On the decidability and complexity of query answering over inconsistent and incomplete databases. In *PODS*, 2003.
- [22] E. Casanovas, P. Dellunde, and R. Jansana. On elementary equivalence for equality-free logic. *Notre Dame Journal of Formal Logic*, 37(3), 1996.
- [23] S. S. Cosmadakis, P. C. Kanellakis, and M. Y. Vardi. Polynomial-time implication problems for unary inclusion dependencies. *JACM*, 37(1), 1990.
- [24] A. Deutsch, B. Ludäscher, and A. Nash. Rewriting queries using views with access patterns under integrity constraints. *TCS*, 371(3), 2007.
- [25] A. Deutsch, L. Popa, and V. Tannen. Query reformulation with constraints. *SIGMOD Record*, 35(1), 2006.
- [26] Facebook. Rate Limiting - Graph API, 2017. <https://developers.facebook.com/docs/graph-api/advanced/rate-limiting/>.
- [27] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: Semantics and query answering. *TCS*, 336(1), 2005.
- [28] W. Fan, F. Geerts, Y. Cao, T. Deng, and P. Lu. Querying big data by accessing small data. In *PODS*, 2015.
- [29] Github. Rate Limit, 2017. <https://developer.github.com/v3/rate-limit/>.
- [30] G. Gottlob, M. Manna, and A. Pieris. Polynomial combined rewritings for existential rules. In *KR*, 2014.
- [31] I. Ileana, B. Cautis, A. Deutsch, and Y. Katsis. Complete yet practical search for minimal query reformulations under constraints. In *SIGMOD*, 2014.
- [32] IMDb. IMDb API, 2017. Compare for instance <http://www.imdb.com/search/title?page=200> and <http://www.imdb.com/search/title?page=201>.
- [33] D. S. Johnson and A. C. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. *JCSS*, 28(1), 1984.
- [34] N. Leone, M. Manna, G. Terracina, and P. Veltri. Efficiently computable Datalog[∃] programs. In *KR*, 2012.
- [35] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *PODS*, 1995.
- [36] C. Li. Computing complete answers to queries in the presence of limited access patterns. *VLDB Journal*, 12(3), 2003.
- [37] C. Li and E. Chang. Answering queries with useful bindings. *TODS*, 26(3), 2001.
- [38] J. C. Mitchell. The implication problem for functional and inclusion dependencies. *Information and Control*, 56(3), 1983.
- [39] A. Nash, L. Segoufin, and V. Vianu. Views and queries: Determinacy and rewriting. *TODS*, 35(3), 2010.
- [40] I. Pratt-Hartmann. Data-complexity of the two-variable fragment with counting quantifiers. *Inf. Comput.*, 207(8), 2009.
- [41] Twitter. API Rate Limits, 2017. <https://dev.twitter.com/rest/public/rate-limiting>.