Querying Data Sources That Export Infinite Sets of Views

Bogdan Cautis
INRIA-Futurs
bogdan.cautis@inria.fr

Alin Deutsch
UC San Diego
deutsch@cs.ucsd.edu

Nicola Onose
UC San Diego
nicola@cs.ucsd.edu

ABSTRACT

We study the problem of querying data sources that accept only a limited set of queries, such as sources accessible by Web services which can implement very large (potentially infinite) families of queries. We revisit a classical setting in which the application queries are conjunctive queries and the source accepts families of (possibly parameterized) conjunctive queries specified as the expansions of a (potentially recursive) Datalog program with parameters. We say that query Q is *expressible* by the program $\mathcal{P}$ if it is equivalent to some expansion of $\mathcal{P}$. Q is *supported* by $\mathcal{P}$ if it has an equivalent rewriting using some finite set of $\mathcal{P}$'s expansions. We present the first study of expressibility and support for sources that satisfy integrity constraints, which is generally the case in practice.

We start by closing a problem left open by prior work even while ignoring constraints, namely the precise relationship between expressibility and support: surprisingly, we show them to be inter-reducible in PTIME in both the absence and the presence of constraints. This enables us to employ the same algorithm for solving both problems.

We identify practically relevant restrictions on the program specifying the views that ensure decidability under a mix of key and weakly acyclic foreign key constraints, and beyond. We show that these restrictions are as permissive as possible, since their slightest relaxation leads to undecidability. We present an algorithm that is guaranteed to be sound when applied to unrestricted input and in addition complete under the restrictions.

As a side-effect of our investigation, we improve the previously best known upper bound for deciding support in the constraint-free case.

1. INTRODUCTION

The recent proliferation of data sources accessible via Web services has renewed interest in the problem of querying sources with restricted querying capabilities [20, 15, 25, 26]. One reason is that, due to commercial, load-control or privacy considerations, Web sources do not typically accept arbitrary application queries against their schema. Instead, they allow only a (potentially infinite) family of (potentially parameterized) queries implemented by the Web services. For instance, the Amazon site provides a service which takes an author name as parameter and returns the corresponding books, but will not allow queries which list all the available books. We refer to the queries accepted by a source as *views*.

In this setting, the application queries issued against the source schema can experience two levels of service. A query can be fully answerable at the source, thus requiring no post-processing at the client. This is the case when the query is equivalent to some view exported by the source (provided the right view can be identified). In many cases, the set of answerable application queries is extended by a *source wrapper* [20], which intercepts them and answers them by automatically identifying a series of relevant views, issuing the corresponding Web service calls and post-processing their results locally.

In this paper, we revisit a classical setting [15, 26] in which the application queries are conjunctive queries and the source accepts families of possibly parameterized conjunctive queries specified as the expansions of a (potentially recursive) Datalog program. The program is said to *generate* these views. As argued in [15, 26] and illustrated below, the choice of Datalog as the view specification formalism enables concise yet expressive descriptions of large sets of views over a given schema. In particular, recursive Datalog programs describe infinitely many views. An additional advantage of Datalog programs besides expressivity is their close relationship to context-free grammars, another natural and universally accepted formalism.

We say that query Q is *expressible* by the program $\mathcal{P}$ if it is equivalent to some view generated by $\mathcal{P}$. Expressible queries can therefore be evaluated at the source, requiring no post-processing at the wrapper. Q is *supported* by $\mathcal{P}$ if it has an equivalent rewriting R using some finite set $\mathcal{V}$ of views generated by $\mathcal{P}$. Note that finding such R and $\mathcal{V}$ witnessing support enables the following execution plan at the wrapper: call the Web services implementing the queries in $\mathcal{V}$, materialize their results locally and run query R over the materialized database.

The challenge in deciding expressibility and support lies in the fact that the family of views can be very large or even infinite. This renders infeasible any systematic enumeration of views. Remarkably, the two problems were previously shown to be decidable [15, 26], however only when ignoring any knowledge of constraints satisfied by the source. In this work, we investigate the effect of source constraints.

The following examples show that source constraints generate new opportunities for detecting support, calling for algorithms which exploit them. (Example 1.1 illustrates a capability-based scenario and will be our running example in this paper. A security scenario is described in Example 1.2.)

EXAMPLE 1.1. *Consider a travel information source conforming to the following schema:*

flight(origin, destination)	*shuttle(origin, destination)*
train(origin, destination)	*bus(origin, destination).*

The source admits only views concerning itineraries by plane, of arbitrary length, such that Paris is reachable by train or bus from the destination airport. This family of views is described as the set of all expansions of the distinguished IDB predicate ans *in program* $\mathcal{P}$ *below:*

$$\begin{aligned} ans(A,B) &\ :-\ \ f(A,C), ind(C,B) \\ ind(C,B) &\ :-\ \ f(C,C'), ind(C',B) \\ ind(C,B) &\ :-\ \ f(C,B), b(B, \text{“}Paris\text{”}) \\ ind(C,B) &\ :-\ \ f(C,B), t(B, \text{“}Paris\text{”}) \end{aligned}$$

Consider an application query that asks for itineraries of length 2 ending in an airport from which Paris is reachable by train, bus and shuttle.

$$\begin{aligned} Q : q(A,B) &\ :-\ \ f(A,C), f(C,B), t(B, \text{“}Paris\text{”}), \\ &\qquad b(B, \text{“}Paris\text{”}), s(B, \text{“}Paris\text{”}) \end{aligned}$$

Clearly, Q is neither expressible nor supported by $\mathcal{P}$ because the views generated by $\mathcal{P}$ do not even mention shuttle information. However, suppose we knew the following constraint to hold on the source (stating that any city pair connected by train and bus is also connected by shuttle):

$$\forall A, S\ t(A,S) \wedge b(A,S) \longrightarrow s(A,S). \tag{1}$$

Then we would like the wrapper to find the rewriting

$$(R) \qquad r(A,B) :- V_1^b(A,B), V_1^t(A,B)$$

where $\{V_i^b\}_{i\geq 1}$ (resp. $\{V_i^t\}_{i\geq 1}$) are families of views generated by $\mathcal{P}$, returning endpoints of itineraries of i flight legs where the destination has a bus link (resp. a train link) to Paris. Indeed, it can be checked that R is equivalent to Q on all databases satisfying (1). Therefore Q is supported by $\mathcal{P}$ when (1) holds.

The following example illustrates an additional motivation for the problem of checking support. For security reasons, sources may allow user access only through *authorized* views which are specified starting from user credentials and access policies of the source [17, 23]. Authorized views may be parameterized. For example, a security policy may require that a physician access a patient record only after providing the corresponding record identifier (see Example 1.2). In the so-called non-Truman access control model [23], a user query is considered legal only if it has an equivalent rewriting based on authorized views, i.e. if it is supported. Illegal queries are rejected by the source.

EXAMPLE 1.2. *Consider a source for medical data, which grants access to patient records only under some conditions. The source conforms to the following schema (where the recordNumber attribute refers to patient visit record number):*

mrecord(patientId, recordNumber)
visit(symptoms, diagnosis, recordNumber)
nextVisit(vID, vID')

and assume that recordNumber is a primary key for the visit relation.

A physician may have access to a limited amount of information concerning patients whose medical records belong to other colleagues, as described by the policy:

“A physician can access the diagnosis for patients only as follows. (1) He can obtain the diagnosis provided he knows the patient identifier and the visit record number. (2) He can also access the diagnosis of visits for patients with symptoms similar to those of a patient whose id and visit record number he knows, as well as of any other follow-up visits. (3) However, the physician can access neither the patient id, nor the visit number for the visits from (2).”

Parts (1) and (2) of the policy could be implemented by separate services, whose authorized views are represented by expansions of distinguished IDB predicate ans_1 *and* ans_2 *respectively in program $\mathcal{P}'$ below (the $?$ annotation denotes parameters):*

$$\begin{aligned} ans_1(S,D) &\ :-\ \ mrecord(?N,?R), visit(S,D,?R), \\ ans_2(D) &\ :-\ \ mrecord(?N,?R), visit(S,D',?R), \\ &\qquad ind_1(S,D,R') \\ ind_1(S,D,R) &\ :-\ \ visit(S,D',R), ind_2(D,R) \\ ind_1(S,D,R) &\ :-\ \ visit(S,D,R) \\ ind_2(D,R) &\ :-\ \ nextVisit(R,R'), ind_2(D,R') \\ ind_2(D,R) &\ :-\ \ visit(S,D,R) \end{aligned}$$

The physician wants to find the symptoms for the visit with record number r_1 of a patient identified by pid_1, together with the diagnosis D_1 for any visit with similar symptoms, and the diagnosis D_2 for a subsequent visit. A conjunctive query that is supported by $\mathcal{P}'$ and provides the information needed is q' below. The primary key constraint is needed to validate the authorization because otherwise there would be no correlation between the information about symptoms used by the views witnessing support.

$$\begin{aligned} q'(S,D_1,D_2) &\ :-\ \ mrecord(pid_1, r_1), visit(S, D_0, r_1), \\ &\qquad visit(S, D_1, R_1), visit(S, D_1', R_1'), \\ &\qquad nextVisit(R_1', R_2), visit(S_2, D_2, R_2) \end{aligned}$$

Note that the system would reject any query trying to retrieve patient ids or visit record numbers, conforming to part (3) of the policy.

Our contributions. In this paper, we carry out the (to the best of our knowledge) first study of the problems of expressibility and support under source constraints. In particular, our list of contributions includes:

Settling an open problem from the constraint-free prior work. We start by closing a problem left open by prior work even while ignoring constraints, namely the precise relationship between expressibility and support: we show them to be inter-reducible in PTIME in both the absence and the presence of constraints. This enables us to employ the same algorithm for solving both problems. The result came as a pleasant surprise, given that in previous work the reported complexity upper bounds for deciding expressivity and support in the constraint-free case were different, suggesting (in line also with intuition) that finding a rewriting of the query using some expansions of the program is harder than finding a single equivalent expansion.

Most permissive restrictions for decidability. We identify practically relevant restrictions on the program which ensure decidability under a mix of key and weakly acyclic foreign key constraints and beyond. These restrictions are particularly useful due to enabling decidability via a reduction to the constraint-free case, which allows one to modularly “plug in” any existing algorithm to this end (such as those in [15, 25, 26] or the one we propose below), using it as a black box. We show that these restrictions are as permissive as possible, since their slightest relaxation leads to undecidability.

A widely-applicable sound test. It is unsatisfactory in practice to refuse to test support and expressibility in the absence of the restrictions implying decidability. A more useful approach consists in devising an algorithm which functions as a decision procedure under these restrictions, yielding only a best-effort “approximation” otherwise. One pragmatic articulation of what “approximation” could mean in this context is the following: the algorithm should be *sound* (i.e. yield no false positives) yet it may return false negatives (i.e. is is not *complete*) for inputs which do not conform to the decidability restrictions. Ruling out false positives is more important than false negatives, as the former lead to returning the wrong answer to the application query, while the latter amount to sometimes rejecting queries even if the source supports them. In this paper we present such an algorithm for both expressibility and support, applicable to arbitrary programs under weakly acyclic sets of embedded dependencies [1], which are sufficiently expressive to capture key and foreign key constraints and beyond. The algorithm runs in deterministic exponential time in the size of the query, the size of the program and the maximum size of a constraint, which is as good as the best deterministic decision algorithm for the case of finitely many views listed individually.

Improved, practically tight upper bounds for the constraint-free problems. As a side-effect of our investigation, we improve the previously best known upper bound for deciding support in the constraint-free case (doubly-exponential time in [15] and non-deterministic EXPTIME in [26] in combined query and program size). The improvement is achieved using the sound algorithm mentioned above, which becomes a decision procedure in the absence of constraints. We show the algorithm to be optimal w.r.t. the program size (we give an EXPTIME lower bound for fixed query) and optimal for practical purposes w.r.t. the query size (we give an NP lower bound for fixed program). The question of the tightness of the NP lower bound in the query size remains open.

Paper outline. After introducing preliminary concepts, results and notation in Section 2, in Section 3 we establish the PTIME inter-reducibility of expressibility and support. We then present in Section 4 restrictions under which we can reduce expressibility and support to the constraint-free case. Section 5 contains a sound algorithm and the improved complexity characterization it enables for the constraint-free problem flavors (Section 5.1). For presentation simplicity, throughout these sections we ignore the presence of parameters in the views generated by the program. We explain parameter handling in Section 6. We give our undecidability results in Section 7. Finally, we discuss related work in Section 8 and conclude in Section 9. Proofs can be found in Appendix A.

2. PRELIMINARIES

We denote with CQ the language of conjunctive queries.

Constraints. We consider constraints ξ of the form

$$\forall \bar{u}\ \phi(\bar{u}, \bar{w}) \longrightarrow \exists \bar{v}\ \psi(\bar{u}, \bar{v})$$

where ϕ and ψ are conjunctions of relational or equality atoms. Such constraints are known as *embedded dependencies* and are sufficiently expressive to specify all usual integrity constraints, such as keys, foreign keys, inclusion, join, multivalued dependencies, etc. [1]. We call ϕ the *premise* and ψ the *conclusion*. If $\bar{v}$ is empty, then ξ is a *full dependency*. If ψ consists only of equality atoms, then ξ is an *equality-generating dependency (EGD)*. If ψ consists only of relational atoms, then ξ is a *tuple-generating dependency (TGD)*. If the premise and conclusion of a TGD contain one atom each, we call it an *inclusion dependency (IND)*. An IND in which the variables $\bar{u}$ appear precisely in the key attributes of the relation mentioned in the conclusion is a *foreign key* constraint. A *key constraint* on relation R can be expressed by the EGD $\forall \bar{u}, \bar{v}_1, \bar{v}_2\ R(\bar{u}, \bar{v}_1) \wedge R(\bar{u}, \bar{v}_2) \longrightarrow \bar{v}_1 = \bar{v}_2$. We write $A \models \mathcal{C}$ if the instance A satisfies all the constraints in $\mathcal{C}$. For brevity, we will refer in the following to embedded dependencies as simply "dependencies".

Containment and Equivalence. Query Q_1 is contained in query Q_2 under the set $\mathcal{C}$ of constraints (denoted $Q_1 \sqsubseteq_{\mathcal{C}} Q_2$) iff $Q_1(D) \subseteq Q_2(D)$ for every database $D \models \mathcal{C}$, where $Q(D)$ denotes the result of Q on D. Q_1 is equivalent to Q_2 under $\mathcal{C}$ (denoted $Q_1 \equiv_{\mathcal{C}} Q_2$) iff $Q_1 \sqsubseteq_{\mathcal{C}} Q_2$ and $Q_2 \sqsubseteq_{\mathcal{C}} Q_1$.

Mappings. A *partial mapping* from CQ query Q_1 to CQ query Q_2 is a function h from the variables and constants of Q_1 to the variables and constants of Q_2 such that (i) h is the identity mapping on all constants, and (ii) for every relational atom (also called subgoal) $R(\bar{X})$ of Q_1, if h is defined for all variables in $(\bar{X})$, then $R(h(\bar{X}))$ is a subgoal of Q_2. A *homomorphism* from a set of subgoals C_1 to a set of subgoals C_2 is a partial mapping from the query $Q_1() :- C_1$ to the query $Q_2() :- C_2$ which is defined on all variables of Q_1. A *containment mapping* from CQ query Q_1 with tuple of head variables $\bar{X}_1$ to CQ query Q_2 with tuple of head variables $\bar{X}_2$ is a homomorphism h from Q_1 to Q_2 such that $h(\bar{X}_1) = \bar{X}_2$. We represent mappings as sets of pairs associating variables with either variables or constants, and use the notation $X : Y$ for the pair (X, Y). The *union* of two mappings is simply the union of their sets of pairs. A mapping is *consistent* if it does not map the same variable to two distinct values. A set of mappings is *compatible* if their union is consistent. Composition of mappings is the standard function composition, denoted with the operator $\circ$.

Expansion using views. Given a CQ query R formulated in terms of a set of view names $\mathcal{V}$, the *expansion* of query R w.r.t. the views in $\mathcal{V}$ (denoted $expand_{\mathcal{V}}(R)$) is the query E obtained as follows: every subgoal $V(\bar{X})$ in R is replaced by a copy of the body of V, in which the head variables of V are renamed to $\bar{X}$ and all other variables are replaced by variables occurring in no other view bodies introduced during the expansion. It is easy to see that this variable renaming defines a homomorphism h from V into the expansion E, which we refer to as the *expansion homomorphism*.

Rewriting using views. We say that CQ query R formulated in terms of view names $\mathcal{V}$ is a rewriting of CQ query Q using $\mathcal{V}$ under set $\mathcal{C}$ of dependencies iff $Q \equiv_{\mathcal{C}} expand_{\mathcal{V}}(R)$.

The chase. We will use the classical *chase* procedure for rewriting conjunctive queries using a set of embedded dependencies [1]. For arbitrary sets $\mathcal{C}$ of dependencies, the chase is not guaranteed to terminate. The least restrictive condition on $\mathcal{C}$ known to date which is sufficient to ensure termination of the chase with $\mathcal{C}$ regardless of the query Q is called *weak acyclicity* [9, 10]. Weak acyclicity of $\mathcal{C}$ implies termination of the chase of Q with $\mathcal{C}$ in time polynomial in the size of Q and exponential in the size of $\mathcal{C}$. The definition of weak acyclicity and the associated result are repeated for the reader's convenience in Appendix C.

Assuming termination of the chase, we denote with $chase_{\mathcal{C}}(Q)$ the query obtained by chasing conjunctive query Q with $\mathcal{C}$ to termination (this query is unique up to equivalence). Besides introducing new variables (for instance due to chasing with TGDs), the chase may equate the original variables of Q to constants or to each other (for instance due to chasing with key constraints) [1]. Denoting this variable renaming with r, it is a well-known fact that r is a homomorphic mapping from Q into $chase_{\mathcal{C}}(Q)$, also called the *chase homomorphism* [1].

Datalog expansions. A finite expansion (in short "expansion") of an IDB predicate p of a Datalog program $\mathcal{P}$ is a CQ query with head $p(\bar{X})$ and body obtained as follows: initialize the body to $body := p(\bar{X})$, then apply the following expansion step a finite number of times until no more IDBs are left in the body: for every IDB goal g_i in the body, pick a rule r_i in $\mathcal{P}$ defining g_i and collect all picked rules in a list $\mathcal{V}$. Treating $\mathcal{V}$ as views, replace *body* with $expand_{\mathcal{V}}(body)$, where each g_i is expanded using r_i. The set of expansions of $\mathcal{P}$ is infinite if $\mathcal{P}$ is recursive.

Convention. *In the remainder of this paper, unless explicitly stated otherwise, all queries and views are conjunctive queries, all programs are Datalog programs, and all dependencies are embedded dependencies.*

3. EXPRESSIBILITY VERSUS SUPPORT

We say that a view V is *generated* by program $\mathcal{P}$ if V is one of the CQ expansions of $\mathcal{P}$.

DEFINITION 3.1. *Given a Datalog program $\mathcal{P}$, a conjunctive query Q and a set of embedded dependencies $\mathcal{C}$, we say that*

1. *Q is* supported *by $\mathcal{P}$ under $\mathcal{C}$ (denoted $\text{SUPP}^{\mathcal{C}}_{\mathcal{P}}(Q)$), iff there is a* finite *set of views $\mathcal{V}$ generated by $\mathcal{P}$ and a conjunctive query rewriting of Q using $\mathcal{V}$ under $\mathcal{C}$.*

2. *Q is* expressible *by $\mathcal{P}$ under $\mathcal{C}$ (denoted $\text{EXPR}^{\mathcal{C}}_{\mathcal{P}}(Q)$), iff Q is equivalent under $\mathcal{C}$ to some view V generated by $\mathcal{P}$.*

In previous work, the problems of expressibility and support were introduced separately (in [15], respectively [26]). They were shown to be decidable, yet their reported complexity upper bounds were different even in the absence of constraints: doubly-exponential deterministic time in [15], improved in [26] to non-deterministic exponential time for support, and EXPTIME for expressibility [26]. These results seemed to suggest that finding a rewriting of the query using some expansions of the program is harder than finding a single equivalent expansion.

In the following, we establish a counter-intuitive relationship between the two problems showing them to be inter-reducible in polynomial time even in the presence of dependencies.

THEOREM 3.1. *Let $\mathcal{C}$ be a weakly acyclic set of embedded dependencies. Then there is a reduction from the problem of support of a query Q by a program $\mathcal{P}$ under $\mathcal{C}$ to that of expressivity under $\mathcal{C}$, which is in PTIME in the size of Q and $\mathcal{P}$ and in EXPTIME in the size of $\mathcal{C}$.*

PROOF. See Appendix A. □

COROLLARY 3.1. *If the schema (with dependencies) is fixed, then there is a PTIME reduction from support to expressibility provided the set of embedded dependencies is weakly acyclic.*

COROLLARY 3.2. *In the absence of dependencies, there is a PTIME reduction from support to expressibility.*

The next result shows the existence of a polynomial-time reduction in the other direction, requiring no restrictions on the embedded dependencies.

THEOREM 3.2. *There is a PTIME reduction from expressibility to support.*

PROOF. See Appendix A. □

4. REDUCING DEPENDENCIES AWAY

In this section, we investigate the conditions under which the problems of expressibility and support can be reduced from the presence of dependencies to the absence thereof. Such reduction immediately yields decidability, as the dependency-free flavors of the problems are known to be decidable [15, 26]. In Section 5, we present a more generally applicable decision procedure which does not rely on reduction to the dependency-free case.

We show in Section 7 that the two problems are in general undecidable, hence it is clear that the targeted reduction does not always exist. Therefore we present a sufficient condition which enables it. This condition turns out to be sufficiently permissive to include practically relevant scenarios: unrestricted programs under weakly acyclic sets of inclusion dependencies (Corollary 4.1), and restricted programs under mixes of key and acyclic sets of inclusion dependencies, in particular keys and foreign keys (Corollary 4.2). Furthermore, as shown in Section 7, the restriction on the program is maximally permissive, in that its relaxation leads to undecidability.

Our reduction relies on the *chase* procedure. This was a natural choice, as the chase tool has been traditionally employed successfully to reduce classical decision problems from the presence of dependencies to their absence[1].

DEFINITION 4.1 ($\mathcal{C}$-INDEPENDENT VIEW SET). *Let $\mathcal{C}$ be a weakly acyclic set of dependencies. We say that a set of views $\mathcal{V} = \{V_1, \ldots, V_n\}$ is $\mathcal{C}$*-independent *iff for every query R formulated in terms of the views,*

$$chase_{\mathcal{C}}(expand_{\{V_1,\ldots,V_n\}}(R))$$

is equivalent even in the absence of dependencies to

$$expand_{\{chase_{\mathcal{C}}(V_1),\ldots,chase_{\mathcal{C}}(V_n)\}}(R).$$

Given a Datalog program $\mathcal{P}$, we denote with $chase_{\mathcal{C}}(\mathcal{P})$ the program obtained by chasing each rule of $\mathcal{P}$ with $\mathcal{C}$.

DEFINITION 4.2 ($\mathcal{C}$-LOCAL PROGRAM). *Let $\mathcal{C}$ be a weakly acyclic set of dependencies. We say that a Datalog program $\mathcal{P}$ is $\mathcal{C}$*-local *iff for every view V generated by $\mathcal{P}$ there is a view W generated by $chase_{\mathcal{C}}(\mathcal{P})$, and for every view W generated by $chase_{\mathcal{C}}(\mathcal{P})$ there is a view V generated by $\mathcal{P}$, such that $chase_{\mathcal{C}}(V)$ is equivalent to W even in the absence of dependencies.*

THEOREM 4.1. *Let Q be a conjunctive query, $\mathcal{C}$ a weakly acyclic set of dependencies, and $\mathcal{P}$ a $\mathcal{C}$-local Datalog program. Then*

(a) $\text{EXPR}^{\mathcal{C}}_{\mathcal{P}}(Q)$ iff $\text{EXPR}^{\emptyset}_{chase_{\mathcal{C}}(\mathcal{P})}(chase_{\mathcal{C}}(Q))$.

(b) If the views generated by $\mathcal{P}$ are $\mathcal{C}$-independent, then $\text{SUPP}^{\mathcal{C}}_{\mathcal{P}}(Q)$ iff $\text{SUPP}^{\emptyset}_{chase_{\mathcal{C}}(\mathcal{P})}(chase_{\mathcal{C}}(Q))$.

We next provide various syntactic restrictions on the dependencies in $\mathcal{C}$ and on $\mathcal{P}$ to guarantee $\mathcal{C}$-independence and $\mathcal{C}$-locality.

THEOREM 4.2. *Let $\mathcal{C}$ be a weakly acyclic set of inclusion dependencies. Then any Datalog program $\mathcal{P}$ is $\mathcal{C}$-local and the views it expresses are $\mathcal{C}$-independent.*

Theorems 4.1 and 4.2 immediately imply that for weakly acyclic sets of inclusion dependencies, expressibility and support reduce to the dependency-free versions:

COROLLARY 4.1. *If $\mathcal{C}$ is a weakly acyclic set of inclusion dependencies, then for any program $\mathcal{P}$ and query Q,*
$\text{EXPR}^{\mathcal{C}}_{\mathcal{P}}(Q)$ iff $\text{EXPR}^{\emptyset}_{chase_{\mathcal{C}}(\mathcal{P})}(chase_{\mathcal{C}}(Q))$ and
$\text{SUPP}^{\mathcal{C}}_{\mathcal{P}}(Q)$ iff $\text{SUPP}^{\emptyset}_{chase_{\mathcal{C}}(\mathcal{P})}(chase_{\mathcal{C}}(Q))$.

We next extend our decidability result to include key constraints. To this end, we require the notion of a program being "key-safe".

Key safety. Let R be a relation with an n-attribute composite key. We say that a rule of $\mathcal{P}$ *outputs the key of R into positions $i_1, \ldots, i_n$* if the sequence of variables $\bar{X}$ located at positions $i_1, \ldots, i_n$ in the rule's head appears in the rule body

[1] Notable examples include equivalence of queries under dependencies, which reduces (provided that the chase terminates) to equivalence of the chased queries in the absence of dependencies [1]; implication of a dependency c by a set $\mathcal{C}$ of dependencies reduces to validity of a dependency obtained by chasing c with $\mathcal{C}$, again assuming chase termination [1].

- either in the key attribute sequence of some R-subgoal, or
- in the positions $j_1, \ldots, j_n$ of some p-subgoal, where p is an IDB predicate with at least one rule that in turn outputs the key of R into the positions $j_1, \ldots, j_n$.

We say that a subgoal g outputs the key of R into the sequence of variables $\bar{X}$ if

- g uses EDB predicate R and $\bar{X}$ appears in the key attributes of g, or
- g uses IDB predicate p and $\bar{X}$ appears in g in the positions into which some rule defining p outputs the key of R.

A rule is safe for the key constraint on R if whenever one of its IDB subgoals outputs the key of R into some sequence of variables $\bar{X}$, no other subgoal does the same. A program $\mathcal{P}$ is key-safe for a set of key constraints $\mathcal{K}$ if each rule is safe for all key constraints in $\mathcal{K}$. Also notice that key-safety can be checked in PTIME in the size of $\mathcal{P}$ and $\mathcal{K}$. If $\mathcal{I}$ is a set of weakly acyclic INDs, $\mathcal{P}$ is key-safe for $\mathcal{C} = \mathcal{K} \cup \mathcal{I}$ if $\mathit{chase}_{\mathcal{I}}(\mathcal{P})$ is key-safe for $\mathcal{K}$.

THEOREM 4.3. *Let $\mathcal{C}$ consist of key constraints and an acyclic set of inclusion dependencies. Any Datalog program $\mathcal{P}$ that is key-safe for $\mathcal{C}$ is also $\mathcal{C}$-local and all views generated by it are $\mathcal{C}$-independent.*

COROLLARY 4.2. *If $\mathcal{C}$ consists of key constraints and an acyclic set of inclusion dependencies and $\mathcal{P}$ is key-safe for $\mathcal{C}$ then for any query Q,*
$\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}}(Q)$ *iff* $\textsc{Expr}^{\emptyset}_{\mathit{chase}_{\mathcal{C}}(\mathcal{P})}(\mathit{chase}_{\mathcal{C}}(Q))$ *and*
$\textsc{Supp}^{\mathcal{C}}_{\mathcal{P}}(Q)$ *iff* $\textsc{Supp}^{\emptyset}_{\mathit{chase}_{\mathcal{C}}(\mathcal{P})}(\mathit{chase}_{\mathcal{C}}(Q))$.

EXAMPLE 4.1. *Consider a source for travel information conforming to the following schema:*

- *train(origin, destination, operator)*
- *bus(origin, destination,operator)*

where each origin-destination pair is connected by a non-stop leg. The source accepts queries for train itineraries with arbitrary many legs, returning the origin, the destination and one intermediary stop. This family of queries is described by program $\mathcal{P}$:

$$\begin{array}{llll}
(\mathcal{P}) & \mathit{ans}(A,B,C) & :- & \mathit{ind}(A,B), \mathit{ind}(B,C) \\
 & \mathit{ind}(B,C) & :- & t(B,B',O), \mathit{ind}(B',C) \\
 & \mathit{ind}(B,C) & :- & t(B,C,O)
\end{array}$$

Let Q be an application query searching for a round-trip with intermediary stop in Paris both on the departure and the return trips, such that between consecutive stops one can always choose between train or bus without changing the travel operator.

$$\begin{array}{llll}
(Q) & q(A,B) & :- & t(A,C,O_1), b(A,C,O_1), \\
 & & & t(C,B,O_2), b(C,B,O_2), \\
 & & & t(B,C,O_3), b(B,C,O_3), \\
 & & & t(C,A,O_4), b(C,A,O_4), \\
 & & & C = \text{“}\mathit{Paris}\text{”}
\end{array}$$

Notice that Q is not supported by $\mathcal{P}$ in the absence of constraints (the source does not even allow views mentioning the bus predicate): $\textsc{Supp}^{\emptyset}_{\mathcal{P}}(Q)$ does not hold.

In contrast, assume next that the source is known to satisfy $\mathcal{C}$, comprised of the inclusion dependency (2) below, which states that every operator will also cover by bus any leg important enough to be covered by train.

$$\forall X, Y, O \; t(X,Y,O) \longrightarrow b(X,Y,O) \qquad (2)$$

Since $\mathcal{C}$ is (trivially) a weakly acyclic set of INDs, by Corollary 4.1 $\textsc{Supp}^{\mathcal{C}}_{\mathcal{P}}(Q)$ holds iff $\textsc{Supp}^{\emptyset}_{\mathit{chase}_{\mathcal{C}}(\mathcal{P})}(\mathit{chase}_{\mathcal{C}}(Q))$ does.

To check the latter, we first note that no chase step with $\mathcal{C}$ applies on Q, so $Q = \mathit{chase}_{\mathcal{C}}(Q)$. However, chase steps do apply on the extensional parts of the second and third rules of $\mathcal{P}$, yielding the new rules (we underline the newly added tuples):

$$\begin{array}{lll}
\mathit{ind}(B,C) & :- & t(B,B',O), \underline{b(B,B',O)}, \mathit{ind}(B',C) \\
\mathit{ind}(B,C) & :- & t(B,C,O), \underline{b(B,C,O)}
\end{array}$$

The new program $\mathit{chase}_{\mathcal{C}}(\mathcal{P})$ generates the family of views V_{ij} denoting the expansion with i legs from the origin to the intermediary point and j legs from the intermediary point to the destination. This includes the view V_{11}:

$$\begin{array}{llll}
(V_{11}) & v(A,B,C) & :- & t(A,B,O_1), b(A,B,O_1), \\
 & & & t(B,C,O_2), b(B,C,O_2)
\end{array}$$

Observe that $\textsc{Supp}^{\emptyset}_{\mathit{chase}_{\mathcal{C}}(\mathcal{P})}(\mathit{chase}_{\mathcal{C}}(Q))$ holds, as witnessed by the equivalent rewriting R of Q using V_{11}:

$$(R) \;\; q(A,B) \; :- \; V_{11}(A, \text{“}\mathit{Paris}\text{”}, B), V_{11}(B, \text{“}\mathit{Paris}\text{”}, A).$$

We conclude that $\textsc{Supp}^{\mathcal{C}}_{\mathcal{P}}(Q)$ holds as well.

5. A WIDELY APPLICABLE TEST

We next present a sound algorithm for testing support, which can be applied to any program and any set of weakly acyclic dependencies. The algorithm is a decision procedure (no false negatives) in all cases when the problem reduces to the dependency-free case (see Section 4), as well as strictly more cases.

Our solution is based on the following overall strategy. Since a systematic enumeration of all (potentially infinitely many) views generated by a program $\mathcal{P}$ is infeasible, we instead “describe the behavior” (in a sense formalized shortly) of any view generated by $\mathcal{P}$ w.r.t. a decision procedure for the existence of a rewriting under $\mathcal{C}$ using finitely many views. This description will abstract away from the view body, focusing on how the view behaves in essential tests performed by the decision procedure. As it will turn out, under our decidability restrictions, there are only *finitely* many distinct behaviors, each exhibited by a possibly infinite set of views. It suffices therefore to find one representative view from each set, thus reducing the problem of checking support by $\mathcal{P}$ to checking the existence of a rewriting using the finitely many representatives. This problem is known to be decidable even in the presence of weakly acyclic dependencies (Lemma 5.1 below). We start by describing the associated decision procedure.

Canonical Rewriting Candidate. Given a finite set of views $\mathcal{V}$, an acyclic set of constraints $\mathcal{C}$, and a query Q, call the *canonical rewriting candidate* of Q using $\mathcal{V}$ under $\mathcal{C}$, denoted $CRC^{\mathcal{C}}_{\mathcal{V}}(Q)$, the query obtained as follows:

(i) it has the same head variables as Q;

(ii) its body is constructed by evaluating each view $V \in \mathcal{V}$ over the body of $\mathit{chase}_{\mathcal{C}}(Q)$ (viewed as a symbolic database, also known as the canonical instance [1]) and adding the subgoal $V(t)$ for every tuple t in the result of the evaluation.

The following reformulates a result in [9] (see also [8]):

LEMMA 5.1 (COROLLARY OF [9]). *Q has a rewriting using $\mathcal{V}$ under $\mathcal{C}$ iff $CRC^{\mathcal{C}}_{\mathcal{V}}(Q)$ is itself one. Moreover, by construction this in turn holds iff* *(a)* $CRC^{\mathcal{C}}_{\mathcal{V}}(Q)$ *is a safe query (its head variables appear in its body), and*

(b) there is a containment mapping from Q *into the result of chasing with* $\mathcal{C}$ *the expansion of* $CRC^{\mathcal{C}}_{\mathcal{V}}(Q)$*:*
$chase_{\mathcal{C}}(expand_{\mathcal{V}}(CRC^{\mathcal{C}}_{\mathcal{V}}(Q))) \sqsubseteq Q$.

EXAMPLE 5.1. *Revisiting Example 1.1, consider the following set of views* $\mathcal{V} = \{V_1, V_2\}$*, generated (among others) by* $\mathcal{P}$*:*

$$\begin{array}{llll}(V_1) & ans^1(Z_1, Z_2) & :- & f(Z_1, X), f(X, Z_2), \\ & & & t(Z_2, \text{“}Paris\text{”}) \\ (V_2) & ans^2(Z_1, Z_2) & :- & f(Z_1, Y), f(Y, Z_2), \\ & & & b(Z_2, \text{“}Paris\text{”})\end{array}$$

By evaluating them on the body of Q *we obtain* $CRC^{\mathcal{C}}_{\mathcal{V}}(Q)$*:*
$R(A, B) :- ans^1(A, B), ans^2(A, B)$*, which is equivalent to* Q *under dependency (1), as can be verified by first constructing the expansion* $E = expand_{\mathcal{V}}(CRC^{\mathcal{C}}_{\mathcal{V}}(Q))$ *as:*

$$\begin{array}{lll}E(A, B) & :- & f(A, X'), f(X', B), t(B, \text{“}Paris\text{”}), \\ & & f(A, Y'), f(Y', B), b(B, \text{“}Paris\text{”})\end{array}$$

which chases with (1) to query (cE):

$$\begin{array}{lll}cE(A, B) & :- & f(A, X'), f(X', B), t(B, \text{“}Paris\text{”}), \\ & & f(A, Y'), f(Y', B), b(B, \text{“}Paris\text{”}), \\ & & \underline{s(B, \text{“}Paris\text{”})}\end{array}$$

into which there is a containment mapping from Q*,* $cm = \{A : A, B : B, C : X'\}$ *(there is another one mapping* C *into* Y'*).*

According to Lemma 5.1, in order for a view to contribute essentially to the rewritability of Q, it (i) must generate a subgoal g of the canonical rewriting candidate; (ii) g's expansion may participate in the chase with $\mathcal{C}$ of the expansion E of the canonical rewriting candidate; and (iii) since Q maps into the chase of E, the expansion of g must include (after the chase) the image of a partial map from Q. We shall therefore describe a view V with respect to its behavior for (i), (ii) and (iii), using the notion of *descriptor*.

Normalized program. For uniformity of treatment, we will assume from now on w.l.o.g. that the program $\mathcal{P}$ is normalized as follows. For every k-ary IDB predicate p, every rule for p has the head variables $\bar{Z} = Z_1, \ldots, Z_k$, in that order. Furthermore, for every EDB predicate e, introduce a new IDB e', replace each occurrence of e in $\mathcal{P}$ with e', and add the rule $e'(\bar{Z}) :- e(\bar{Z})$. The normalized program has only two kinds of rules: those whose bodies consist of a single EDB subgoal (called *EDB rules*), or solely of IDB subgoals (called *IDB rules*). For technical reasons, we additionally compute (as in [15]), the *closure* of the program, which consists in adding for every rule r in $\mathcal{P}$ all rules obtained from r by systematically equating in all possible ways the head variables of r with each other and with the constants in Q.

DEFINITION 5.1 (DESCRIPTORS). *Given an IDB predicate* p *and a conjunctive query body* E *over EDBs from* $\mathcal{P}$*,* $E^{(p(t), fr)}$ *is a* descriptor *iff* $\mathcal{P}$ *generates as expansion of* p *a query of head variables* $\bar{Z}$*,* $p(\bar{Z}) :- body$*, such that*

- *there is a homomorphism* $to : body \to chase_{\mathcal{C}}(Q)$ *s.t.* $to(\bar{Z}) = t$*; and*
- fr *is a partial variable mapping from* Q *into* $chase_{\mathcal{C}}(body)$ *such that the image of* Q *under* fr *is* E*.*

We call E *the* expansion fragment *described by the descriptor, and* $(p(t), fr)$ *the* adornment *of* E*. We call variables* $\{Z_1, \ldots, Z_k\}$ *(where* k *is the arity of* p*) the* distinguished *variables of the descriptor, while all other variables in the range of* fr *are* hidden.

EXAMPLE 5.2. *In the setting of Example 5.1,* $d_1 = E_1^{(p_1(t_1), fr_1)}$ *and* $d_2 = E_2^{(p_2(t_2), fr_2)}$ *below are descriptors for the views* V_1 *and* V_2*, respectively:*

$$\begin{array}{rcl}E_1 & = & [f(Z_1, X), f(X, Z_2), t(Z_2, \text{“}Paris\text{”})] \\ p_1(t_1) & = & ans(A, B) \\ fr_1 & = & \{A : Z_1, C : X, B : Z_2\} \\ E_2 & = & [b(Z_2, \text{“}Paris\text{”})] \\ p_2(t_2) & = & ans(A, B) \\ fr_2 & = & \{B : Z_2\}\end{array}$$

Note that, though the two views contribute the same $ans(A, B)$ *goal to the canonical rewriting candidate, the two descriptors distinguish among them by the images of* Q *into the view bodies (*E_1 *includes the image of* Q*'s* t *and two* f *goals,* E_2 *only that of the* b *goal).*

Before explaining in detail how descriptors are found, we show how they can be used to soundly infer support.

Intuitively, a descriptor represents the fragment of a (chased) view generated by $\mathcal{P}$, which serves as the image of the partial mapping from Q. Our goal is to put together such fragments in a consistent way to create (if it exists) the image of Q under a *containment mapping*.

Partial rewriting candidate. More formally, consider a finite set of descriptors w.r.t. to query Q, program $\mathcal{P}$ and dependencies $\mathcal{C}$: $\mathcal{D} = \{E_i^{(p_i(t_i), fr_i)}\}_{1 \le i \le n}$, where all p_i are (not necessarily distinct) distinguished IDBs of $\mathcal{P}$. Introduce for each predicate p_i a fresh predicate p_i^i (using the rank i of the predicate in an arbitrary ordering of the descriptor set) such that $p_i^i \neq p_j^j$ for all $1 \le i, j \le n$. Assuming w.l.o.g. that Q's tuple of head variables is $\bar{X}$, we call the query $R(\bar{X}) :- p_1^1(t_1), \ldots, p_n^n(t_n)$ the *partial rewriting candidate* described by $\mathcal{D}$. The set $\mathcal{V} := \{VF_i : p_i^i(\bar{Z}) :- E_i\}_{1 \le i \le n}$ is called the *view fragments* described by $\mathcal{D}$. The view fragments VF_i are not necessarily safe queries, if not all the head variables serve as image of the partial mapping fr_i.

EXAMPLE 5.3. *For the set of descriptors* $\mathcal{D} = \{d_1, d_2\}$ *from Example 5.2, the fresh view goals are* ans^1, ans^2 *respectively. The partial rewriting candidate described by* $\mathcal{D}$ *is*
$R(A, B) :- ans^1(A, B), ans^2(A, B)$ *(it happens to coincide with the canonical rewriting candidate shown in Example 5.1). The view fragments are*

$$\begin{array}{llll}(VF_1) & ans^1(Z_1, Z_2) & :- & f(Z_1, X), f(X, Z_2), t(Z_2, \text{“}Paris\text{”}) \\ (VF_2) & ans^2(Z_1, Z_2) & :- & b(Z_2, \text{“}Paris\text{”}).\end{array}$$

Notice how VF_1, VF_2*'s bodies are isomorphic to fragments of the bodies of views* V_1*, respectively* V_2 *from Example 5.1. Also notice that* VF_2 *is not a safe query as variable* Z_1 *does not appear in the body.*

The following result allows us to test support, as in Lemma 5.1, but using descriptors instead of explicit views. The key idea is to use the partial rewriting candidate instead of the canonical rewriting candidate ($CRC^{\mathcal{C}}_{\mathcal{V}}(Q)$).

COROLLARY 5.1 (OF LEMMA 5.1). *Let* $\mathcal{D}$ *be a finite set of descriptors w.r.t. query* Q*, program* $\mathcal{P}$ *and dependencies* $\mathcal{C}$*:*
$\mathcal{D} = \{E_i^{(p_i(t_i), fr_i)}\}_{1 \le i \le n}$*. Denote with*

- R *the partial rewriting candidate described by* $\mathcal{D}$*;*
- $\mathcal{V}$ *the view fragments described by* $\mathcal{D}$*;*
- E *the expansion* $expand_{\mathcal{V}}(R)$

If

(a) R is safe and

(b) there exists a containment mapping cfr from Q into $chase_{\mathcal{C}}(E)$,

then Q is supported by $\mathcal{P}$ under $\mathcal{C}$.

We say that any set $\mathcal{D}$ as in Corollary 5.1 *witnesses support.* Notice that conditions (a) and (b) in Corollary 5.1 reformulate the corresponding conditions from Lemma 5.1 in terms of descriptors.

EXAMPLE 5.4. *The set of descriptors $\mathcal{D}$ in Example 5.3 witnesses support for the query, program and dependency in our running Example 1.1. Indeed, if we apply the test of Corollary 5.1 to the partial rewriting candidate R and the view fragments VF_1 and VF_2 described by $\mathcal{D}$(shown in Example 5.3), we obtain*

- *the expansion*

$$\begin{aligned} EF(A, B) \quad :- \quad & f(A, X'), f(X', B), t(B, \text{“Paris”}), \\ & b(B, \text{“Paris”}) \end{aligned}$$

- *the result (cEF) of chasing EF with dependency (1),*

$$\begin{aligned} cEF(A, B) \quad :- \quad & f(A, X'), f(X', B), t(B, \text{“Paris”}), \\ & b(B, \text{“Paris”}), \underline{s(B, \text{“Paris”})}; \end{aligned}$$

Notice that EF and cEF are fragments of E, respectively cE from Example 5.1. Let cfr be the mapping $cfr = \{A : A, B : B, C : X'\}$. We observe that (a) R is safe; and (b) cfr is a containment mapping from Q into cEF, thus satisfying the conditions of Corollary 5.1.

Clearly, the number of descriptors is infinite due to the unbounded set of hidden variables, but there are only finitely many isomorphism types of descriptors modulo renaming of the hidden variables, in the following sense:

DEFINITION 5.2 (SIMILARITY). *Two descriptors $E_1^{(p_1(t_1),fr_1)}$ and $E_2^{(p_2(t_2),fr_2)}$ are* similar *iff $p_1 = p_2$ (and hence the distinguished variables of the descriptors are the same), $t_1 = t_2$, and there is an isomorphism i between the ranges of fr_1 and fr_2 which is the identity on the distinguished variables, and i witnesses the isomorphism of E_1 and E_2.*

Intuitively, the condition on fr_1 and fr_2 enforced by similarity ensures that the partial containment mapping of Corollary 5.1, restricted to the view fragment, is the same for both descriptors.

It is easy to see that similarity is an equivalence relation, and that there are only finitely many equivalence classes of descriptors under similarity. Indeed in $E^{(p(t),fr)}$, p is a predicate from $\mathcal{P}$; t a tuple of variables and constants from $chase_{\mathcal{C}}(Q)$, thus the number of distinct values it can take is polynomial in the size of $chase_{\mathcal{C}}(Q)$ and exponential in the arity of p; the number of distinct (up to isomorphism) partial mappings *fr* is exponential in the number of variables in Q.

Similarity plays a key role in our support test. Indeed we can show that any representative of a similarity equivalence class is as good as any member of the class for the purpose of witnessing support, in the following sense:

(†) if descriptor d_1 is similar to d_2, then for any set $\mathcal{D}$ of descriptors, $\mathcal{D} \cup \{d_1\}$ is a support witness if and only if $\mathcal{D} \cup \{d_2\}$ is.

Algorithm findDescriptors. We next present a bottom-up algorithm for computing representatives of descriptor equivalence classes under similarity. The algorithm **findDescriptors** consists in initializing a set of descriptors $\mathcal{D}$ to the empty set, then repeatedly carrying out the rule steps described below until $\mathcal{D}$ reaches a fixpoint (under similarity), finally returning $\mathcal{D}$.

EDB rule step. Consider an EDB rule

$$e'(Z_1, \ldots, Z_k) :- e(Z_1, \ldots, Z_k).$$

For every variable mapping to from $Z_1, \ldots, Z_k$ into Q's variables and constants, such that the goal $e(to(Z_1), \ldots, to(Z_k))$ appears in $chase_{\mathcal{C}}(Q)$; and every partial variable mapping *fr* from the variables of Q to $\{Z_1, \ldots, Z_k\}$ (including the empty-domain one), add to $\mathcal{D}$ the descriptor $E^{(e(to(\bar{Z})),fr)}$, where $E = e(\bar{Z})$. Note that descriptors with empty-domain mappings capture the situation when none of the query goals maps into the described e goal[2].

IDB rule step. Consider an IDB rule

$$p(\bar{X}) :- p_1(\bar{X}_1), \ldots, p_n(\bar{X}_n).$$

If there exists a homomorphism h from the rule body into $chase_{\mathcal{C}}(Q)$, and a set of descriptors

$$E_1^{(p_1(h(\bar{X}_1)),fr_1)}, \ldots, E_n^{(p_n(h(\bar{X}_n)),fr_n)}$$

in $\mathcal{D}$, then:

Construct the views $V_i : p_i(\bar{Z}_i) :- E_i$. Denote with E the expansion of the rule body using these views, and with xh_i the corresponding expansion homomorphism $xh_i : E_i \rightarrow E$ (i.e. the variable renaming performed on each V_i during expansion). Chase E with $\mathcal{C}$ and denote with ch the corresponding chase homomorphism $ch : E \rightarrow chase_{\mathcal{C}}(E)$. If the set $\{ch \circ xh_i \circ fr_i\}_{1 \leq i \leq n}$ of partial mappings from Q into $chase_{\mathcal{C}}(E)$ is compatible, construct the combined mapping $cfr := \bigcup_{i=1}^{n} ch \circ xh_i \circ fr_i$, otherwise exit the rule step.

For every partial mapping *fr* from Q into $chase_{\mathcal{C}}(E)$ which extends *cfr* (including the trivial extension $fr = cfr$) by mapping additional variables of Q into fresh variables added during the chase, compute descriptor $d = F^{(p(h(\bar{X})),fr)}$, where F is the image under *fr* of all goals in Q s.t. *fr* is defined on all their variables. If d is not similar to any descriptor in $\mathcal{D}$, add it to $\mathcal{D}$.

EXAMPLE 5.5. *We next illustrate the rule steps of algorithm* **findDescriptors** *for Example 1.1 showing how descriptors d_1 and d_2 from Example 5.2 are derived. First, observe that no chase step applies on Q, so $Q = chase_{\mathcal{C}}(Q)$.*

For brevity, we work on the unnormalized program $\mathcal{P}$. Applications of EDB rule steps produce (among others) the following descriptors:

$$\begin{aligned}
&(d_3) && [f(Z_1, Z_2)]^{(f(A,C),\{A:Z_1,C:Z_2\})} \\
&(d_4) && [f(Z_1, Z_2)]^{(f(A,C),\{\})} \\
&(d_5) && [f(Z_1, Z_2)]^{(f(C,B),\{C:Z_1,B:Z_2\})} \\
&(d_6) && [f(Z_1, Z_2)]^{(f(C,B),\{\})} \\
&(d_7) && [t(Z_1, \text{“}Paris\text{”})]^{(t(B,\text{“}Paris\text{”}),\{B:Z_1\})} \\
&(d_8) && [b(Z_1, \text{“}Paris\text{”})]^{(b(B,\text{“}Paris\text{”}),\{B:Z_1\})}
\end{aligned}$$

Notice that for the same match of EDB goal $f(Z_1, Z_2)$ into goal $f(A, B)$ of $chase_{\mathcal{C}}(Q)$, several partial mappings from the query

[2] Technically, descriptors for EDB rule IDBs using empty-domain partial mappings do not fully conform to Definition 5.1 as the expansion fragment contains a goal that is not the image under the partial mapping. As seen in the IDB rule step, the definition holds for all other IDBs, which are the pre-normalization IDBs.

are considered. We show only two here (in descriptors d_3 and d_4, where the latter uses the empty mapping, meaning that no query variable is mapped into its fragment).
An IDB rule step for the fourth $\mathcal{P}$ rule combines descriptors d_5 and d_7 yielding a new one:

$$(d_9)\quad [f(Z_1, Z_2), t(Z_2, \text{“}Paris\text{”})]^{(ind(C,B),\{C:Z_1,B:Z_2\})}$$

which combines with d_3 using the first rule of $\mathcal{P}$, yielding d_1.

Descriptors d_6 and d_8 combine via an IDB rule step with the third rule in $\mathcal{P}$ to

$$(d_{10})\quad [b(Z_2, \text{“}Paris\text{”})]^{(ind(C,B),\{B:Z_2\})}$$

which combines with d_4 using the first rule of $\mathcal{P}$, yielding d_2.

The following result guarantees that the inflationary process for descriptor discovery implemented by algorithm **findDescriptors** terminates for weakly acyclic sets of constraints.

LEMMA 5.2. *If $\mathcal{C}$ is weakly acyclic, then algorithm* **findDescriptors** *is guaranteed to*

(a) *terminate in time exponential in the sizes of $\mathcal{P}$, $\mathcal{C}$, and Q.*

(b) *output only (pairwise dissimilar) descriptors.*

PROOF. See Appendix A. □

Algorithm testSupport. Our algorithm for testing support amounts to deciding if the descriptors computed by algorithm **findDescriptors** give a support witness (in the sense of Corollary 5.1). According to Corollary 5.1, the existence of such a witness is sufficient for support, but, due to our undecidability results, when the program is unrestricted (see Section 7), it is not always a necessary condition. That is why algorithm *testSupport* is in general only sound.

```
algorithm testSupport
input: query $Q$, program $\mathcal{P}$, set of dependencies $\mathcal{C}$;

begin
  $\mathcal{D} := \mathbf{findDescriptors}(Q, \mathcal{P}, \mathcal{C})$;
  $\mathcal{D}'$ := all descriptors from $\mathcal{D}$ pertaining to distinguished predicates of $\mathcal{P}$;
  if $\mathcal{D}'$ witnesses support (tested as in in Corollary 5.1)
     then return true;
     else return false;
end
```

Algorithm **testSupport** enjoys the following properties.

THEOREM 5.1. *If $\mathcal{C}$ is weakly acyclic, the following hold:*

1. *Algorithm* **testSupport** *is sound for testing support.*
2. *Algorithm* **testSupport** *runs in time exponential in the size of $\mathcal{P}$, $\mathcal{C}$, and Q.*

PROOF. (1.) follows from Lemma 5.2(b) and Corollary 5.1.
(2.) follows from Lemma 5.2(a) and Corollary 5.1, noticing that the containment mapping *cfr* can be computed in EXPTIME in the size of Q and in PTIME in the size of the result of chasing the partial rewriting candidate. In turn, the size of the chase result is exponential in the maximum arity of a constraint in $\mathcal{C}$ and polynomial in that of the partial rewriting candidate [9]. The size of the partial rewriting candidate is given by the maximum number of distinct descriptors that can be built, which by Lemma 5.2(a) is worst-case exponential in the size of Q, $\mathcal{C}$, and $\mathcal{P}$. □

Algorithm **testSupport** produces strictly less false negatives than the approach of reducing away dependencies described in Section 4. First, it is a decision procedure whenever the reduction succeeds:

THEOREM 5.2. *If $\mathcal{C}$ is weakly acyclic and $\mathcal{P}$ is a $\mathcal{C}$-local program generating $\mathcal{C}$-independent views, then algorithm* **testSupport** *is a decision procedure for support.*

COROLLARY 5.2. *If $\mathcal{C}$ is a weakly acyclic set of key and foreign key constraints, and $chase_{\mathcal{C}}(P)$ is safe for the keys in $\mathcal{C}$, then* **testSupport** *is a decision procedure for support.*

Second, the setting of Example 1.1 exhibits a case in which the restrictions required in Section 4 for reduction to the dependency-free case do not apply (they involve keys and foreign keys, while dependency (1) is neither). Indeed, it is easy to check that the chased program does not support the chased query in the absence of dependencies. We therefore need a qualitatively better approach, which is provided by algorithm **testSupport**: Example 5.5 shows that the call to **findDescriptors** yields (among others) the descriptors d_1, d_2, which, according to Example 5.4, witness support.

Algorithm testExpressibility. While we could use the reduction from expressibility to support provided in Theorem 3.2, the following minor variation on algorithm **testSupport** constitutes a direct test: call algorithm **findDescriptors**, keep only the descriptors for distinguished IDB predicates, and perform the test of Corollary 5.1 only on singleton sets of descriptors.

Finding the actual views. So far, we have only provided algorithms for deciding support and expressibility. To turn them into algorithms exhibiting the actual views generated by $\mathcal{P}$ as well as the rewriting using it requires extra bookkeeping. All we need to do is to carry along with a descriptor d the actual expansion built during its derivation, noticing that the derivation tree of d coincides with the expansion tree of the expansion described by d.

Finding minimized witnesses for support. Let us note that while the partial rewriting candidate described by $\mathcal{D}'$ in algorithm **testSupport** may contain redundant atoms, in security applications we only need to check if a query is supported by authorized views [23], which amounts to checking the existence of a rewriting without ever using it. Instead, the original query is executed once it is authorized. For non-security applications in which the wrapper needs to find and execute the rewriting in order to service a user application, one can plug in any technique for minimization under constraints already studied in the literature. For instance, the backchase minimization [8] which starts from the rewriting candidate (corresponding to R from Corollary 5.1) and considers subsets of view atoms at a time. This technique is amenable to further optimization by reusing the information from the partial mappings stored in the descriptors: find subsets of descriptors whose partial mappings are compatible and yield a total mapping from the query into the partial rewriting candidate. The presentation of such an optimization algorithm combining the discovered descriptors more efficiently goes beyond the scope of this paper.

5.1 Revisiting the Dependency-free Case

Based on algorithm **testSupport**, we now improve the previously best-known upper bound for checking support in the dependency-free setting. [15] reported a deterministic doubly-exponential upper bound in the size of the query and program, while [26] improved it to non-deterministic exponential time.

First, we observe that Theorem 5.2 and Theorem 5.1 imply the following upper bound:

COROLLARY 5.3. *In the absence of dependencies, algorithm* **testSupport**

(a) *is a decision procedure for support of a query by an arbitrary program, and* *(b) runs in deterministic exponential time in the sizes of the program and query.*

We next show that this upper bound is tight in the program size, and tight for practical purposes in the query size.

THEOREM 5.3. $\text{SUPP}^{\emptyset}_{\mathcal{P}}(Q)$ *is NP-hard in the size of* Q *and EXPTIME-complete in the size of* $\mathcal{P}$.

PROOF. See Appendix A. □

6. PARAMETERS

In this section, we show how our approach on expressibility and support can be extended to the case when sources implement parameterized queries, expecting applications to provide the parameter values. In other words, a parameter will denote an *input* variable, that has to be *bound* to a value before the view can be evaluated. Head variables will be called *output* variables.

Notation. For parameters we adopt the $?X$ notation of [14, 15], enabling the generation of parameterized views. We stress that by this notation, an input variable $?X$ will be considered different from some other variable X appearing in the same program rule.

EXAMPLE 6.1. *Consider the following program which generates views outputting destination airports with bus and train connection to Paris, provided that the source and intermediary airports are specified by the application:*

$$\begin{array}{rcl} ans(B) & :- & f(?A, C), ind(C, B) \\ ind(?C, B) & :- & f(?C, C), ind(C, B) \\ ind(?C, B) & :- & f(?C, B), b(B, \text{“}Paris\text{”}) \\ ind(?C, B) & :- & f(?C, B), t(B, \text{“}Paris\text{”}) \end{array}$$

Notice that while the number of output variables is fixed (it is the arity of the distinguished predicates), the program imposes no bound on the number of input variables of generated views. In practice, this corresponds to describing service calls with a variable number of arguments.

There are two kinds of query evaluation plans one may adopt in the presence of parameters. The straightforward execution consists in the wrapper issuing in a first stage a series of service calls to the source without inspecting any intermediate results to determine how to instantiate parameters for the other calls. Once all call results come in, during the second stage the rewriting query is run over them and the result passed to the application query. This is the approach taken in [14, 15]. We shall call this approach the *two-stage* evaluation. The drawback of this approach is that it does not explore plans in which output values of one view goal become the input to another view goal in the rewriting. A more sophisticated evaluation strategy is based on this idea, of interleaving query execution at the wrapper with service calls to the source. The evaluation of a subquery of the rewriting can thus provide parameter values for the subsequent calls needed by the non-evaluated part of the rewriting. This approach is used in [25] and, for finite sets of parameterized views, in [11], where it is known as the *dependent-join* evaluation. Section 6.1 discusses two-stage evaluation briefly and Section 6.2 details expressibility and support under the more advanced evaluation strategy.

6.1 Two-stage evaluation

If only two-stage evaluation is considered, there is an immediate reduction to the problem of non-parameterized views, based on the following observation:

LEMMA 6.1. *In two-stage evaluation, for the views to be relevant to the problem of support or expressibility, their parameters must be filled in with constants appearing in the query or the source dependencies.*

This result follows immediately from Lemma 5.1 and generalizes a similar observation from [14] to the presence of dependencies. It implies that it suffices to generate a new program in which the parameters are replaced in all possible ways by the (bounded) set of constants in Q and $\mathcal{C}$, and test support and expressibility for the new program. In practice, an efficient implementation would extend the rewriting algorithm as suggested in [14], by mapping parameters into constants from the query.

6.2 Dependent-join evaluation

We next discuss the second, more flexible approach, which uses dependent-join evaluation plans. We start by introducing some auxiliary notions.

Access patterns. An access pattern for a view $V(X_1, \ldots, X_k)$ is an expression α in $\{o, i\}^k$. We say that the X_j is an *output* (resp. *input*) variable if $\alpha(j) = o$ (resp. $\alpha(j) = i$). A view with access pattern α is denoted $V^{\alpha}(X_1, \ldots, X_k)$. Views generated by a Datalog program with parameters will be presented using this notation, by introducing an input head variable for each parameter.

Executable query. Following [19, 7], we say that a query R formulated in terms of view names with binding patterns $\mathcal{V}$ is *executable* if the access patterns of R are such that every input variable appears first in an output position of some previous goal.

Expressibility / Support. We are now ready to extend the definitions of expressibility and support in the presence of parameterized views. We say that a query Q is *expressible* by a program $\mathcal{P}$ iff the query is equivalent to a query obtained from an expansion of $\mathcal{P}$ by replacing all input variables by constants. Note that this is the natural choice, since expressibility captures the cases in which a query can be fully answered by just one “service call”, without any post-processing. We say that Q is *supported* by $\mathcal{P}$ iff there exists an executable rewriting R using some finite set $\mathcal{V}$ of views with access patterns generated by $\mathcal{P}$.

Before going into the specific details, we first give a brief outline on how the solutions of the previous section can be extended to deal with parameterized programs. As before, we aim at reducing these problems to query answering using only a finite family of the specified views, defined by descriptors. First, since by the dependent-join mechanism input variables play an important role in how view goals interact in a rewriting, we need to keep track in descriptors of their query-view and view-query mappings. While this leads to descriptors of unbounded size (since the number of input variables is not bounded) we show that only a finite set of descriptors needs to be considered. Then, we extend algorithm **testSupport** to find an *executable* ordering of a rewriting in terms of descriptors. For this phase, we show that an expensive ordering search can be avoided, by relying on a canonical executable rewriting candidate. In conclusion, similar to the case without parameters, we obtain a sound, exponential-time, algorithm for expressibility and support, which becomes complete in the absence of constraints or under restrictions on the interaction between program and constraints.

Modifying Example 1.1, the running example in this section is the following:

EXAMPLE 6.2. *Consider the schema from Example 1.1 extended with a relation* airport(name) *and the set of views specified by the parameterized Datalog program* $\mathcal{P}''$*, with 2 distinguished idb predicates (*ans_1 *and* ans_2*):*

$$\begin{array}{rcl} ans_1(A) & :- & a(A) \\ ans_2(A,B) & :- & f(A,C), ind(C,B) \\ ind(C,B) & :- & f(C,C'), ind(C',B) \\ ind(C,?B) & :- & f(C,?B), b(?B, \text{“Paris”}) \\ ind(C,?B) & :- & f(C,?B), t(?B, \text{“Paris”}) \end{array}$$

Note that the program differs from the one of Example 1.1 in two aspects: (a) the source admits direct access to the airport *relation (by ans_1) and (b) the destination of views concerning itineraries (by ans_2) is an input variable.*

Besides dependency (1)

$$\forall A, S\ t(A,S) \wedge b(A,S) \longrightarrow s(A,S)$$

we assume the source verifies also the dependency

$$\forall A, b(A, \text{“}Paris\text{”}), t(A, \text{“}Paris\text{”}) \longrightarrow a(A) \qquad (3)$$

which guarantees that any airport with a bus and train connection to Paris can be found in the airport *relation.*

Consider that the user asks the same query as in Example 1.1, i.e., itineraries of length 2 ending in an airport from which Paris is reachable by all the three transportation means

$$\begin{array}{rcl} Q : q(A,B) & :- & f(A,C), f(C,B), t(B, \text{“Paris”}), \\ & & b(B, \text{“}Paris\text{”}), s(B, \text{“Paris”}). \end{array}$$

We recall that this query was supported in the setting of Example 5.1, as witnessed by the rewriting

$$(R) \qquad r(A,B) :- \ V_1(A,B), V_2(A,B).$$

However, under the given access patterns, R is no longer a witness for support since, when considering access patterns, the conjunction of $V_1^{oi}(A,B)$ and $V_2^{oi}(A,B)$ is not executable. But by adding to this rewriting $U^o(B)$ as a first subgoal, where U is the one view generated by predicate ans_1, the query becomes executable:

$$(R') \qquad r(A,B) :- \ U^o(B), V_1^{oi}(A,B), V_2^{oi}(A,B),$$

and moreover equivalent to Q. Indeed, the values for B are now passed by the dependent-join, and it can be easily checked that R' is equivalent to Q under the two dependencies, since the airport *goal of the rewriting maps in the result of chasing Q with (3).*

We next discuss the decision procedure for view-based query answering using a *finite* set of parameterized views, under dependencies. This procedure will be then adapted to a finite set of view descriptors.

Answerable part. Given a query R formulated in terms of view names with access patterns $\mathcal{V}$, we call the *answerable part* of R (denoted $ans(R)$) the executable query with the same head as R and the body built one goal at a time from $body(R)$ as follows:

- start with an empty set of bounded variables, $\mathcal{B}$, then repeatedly
- find the first view goal $g^\alpha(\bar{X})$ in R not added to $ans(R)$ such that all the input variables of g are in $\mathcal{B}$; add this goal to $ans(R)$ and add its head variables $\bar{X}$ to $\mathcal{B}$.

Clearly, $ans(R)$ is an executable query and this procedure runs in quadratic time.

Executable Canonical Rewriting Candidate. Given a finite set of views with access patterns $\mathcal{V}$, an acyclic set of constraints $\mathcal{C}$, and a query Q, we call the *executable canonical rewriting candidate* of Q using $\mathcal{V}$ under $\mathcal{C}$, denoted $ECRC_{\mathcal{V}}^{\mathcal{C}}(Q)$, the query obtained as follows:

(i) compute $CRC_{\mathcal{V}}^{\mathcal{C}}(Q)$ (as described in Section 5),

(ii) find its answerable part, $\mathbf{ans}(CRC_{\mathcal{V}}^{\mathcal{C}}(Q))$.

Similar to Lemma 5.1, results from [7] guarantee that Q has a rewriting using $\mathcal{V}$ under $\mathcal{C}$ iff $ECRC_{\mathcal{V}}^{\mathcal{C}}(Q)$ is itself one. We omit further details and only illustrate the main idea by an example.

EXAMPLE 6.3. *Revisiting Example 6.2, we know that the views V_1, V_2, U, generated (among others) by $\mathcal{P}''$, give an executable rewriting for Q, under $\mathcal{C} = \{(1),(3)\}$.*

Assume that an additional distinguished predicate is present in $\mathcal{P}''$, defined by the rule:

$$ans_3(A,B) \quad :- \quad f(A,?C), ind(?C,B)$$

This rule generates, among others, two views that have the same subgoals as V_1 and V_2, but in which the intermediary stop is an input variable, too. Hence these views, denoted W_1 and W_2, will have one output and two inputs, their access pattern being (o,i,i).

Consider the set of views $\mathcal{V} = \{V_1, V_2, U, W_1, W_2\}$, which can all be mapped into $chase_{\mathcal{C}}(Q)$. By evaluating them on the body of $chase_{\mathcal{C}}(Q)$, we obtain the intermediary $CRC_{\mathcal{V}}^{\mathcal{C}}(Q)$:

$$\begin{array}{rcl} R(A,B) & :- & V_1^{oi}(A,B), V_2^{oi}(A,B), U^o(B), \\ & & W_1^{oii}(A,C,B), W_2^{oii}(A,C,B) \end{array}$$

which is not executable since no value can be assigned to the C input variable. However, by computing $\mathbf{ans}(CRC_{\mathcal{V}}^{\mathcal{C}}(Q))$, we eliminate the last two goals and reorder the rewriting, obtaining the $ECRC_{\mathcal{V}}^{\mathcal{C}}(Q)$:

$$R(A,B) \quad :- \quad U^o(B), V_1^{oi}(A,B), V_2^{oi}(A,B)$$

which we know is indeed an executable rewriting of Q.

Testing expressibility and support. Similarly to the case without access patterns, we capture the usefulness of a view in the executable rewriting candidate by a descriptor, which takes also into account parameters and the access patterns they impose. Once the set of descriptors is obtained, checking expressibility amounts to checking if one of them denotes a view which becomes equivalent to Q after replacing input variables by constants. For testing support, we first construct the *partial rewriting candidate*, as described in Section 5. Since this candidate may not be executable, we need to compute its answerable part, which we call the *executable partial rewriting candidate*. Finally, we check as in Corollary 5.1 whether this candidate is equivalent to Q under the dependencies, starting from the corresponding view fragments.

Finding descriptors Since now we need to describe also the role of view goals in the answerable part of the rewriting candidate, we enrich the descriptor definition by taking into account input variables. More precisely, (a) input variables are treated as head variables, and (b) we add the corresponding access patterns to each descriptor, thus discriminating among views which are similar according to Definition 5.2 if they have distinct access patterns. For space reasons, we omit the formal definition and illustrate these changes on the setting of Example 6.3:

EXAMPLE 6.4. *The descriptor for the view V_1^{oi} has besides the components given in Example 5.2 the access pattern $\alpha_1 = (o,i)$. Similarly, the descriptor for the view W_1^{oii} has the components*

$$\begin{array}{rcl} E_1 & = & [f(Z_1,Z_2), f(Z_2,Z_3), t(Z_3, \text{“}Paris\text{”})] \\ p_1(t_1) & = & ans(A,C,B) \\ fr_1 & = & \{A : Z_1, C : Z_2, B : Z_3\} \\ \alpha_2 & = & (o,i,i) \end{array}$$

One difficulty in extending descriptors in this way comes from the fact that there may be no bound on the number of input variables of generated views, leading to an unbounded number of descriptors and excluding any rewriting approach based on descriptors. However, we know from [21] that, *in the absence of constraints*, if a rewriting using views with binding patterns exists, then one with at most n (the number of variables of Q) distinct variables exists. This can in fact be extended to the case *with constraints*, showing that if a rewriting with a finite set of views exists, then there is also one in which the view atoms have at most n input variables, n being the number of variables of $chase_{\mathcal{C}}(Q)$. The intuition for this is that if a view with more than n parameters appears in a rewriting, then for sure some of those parameters will be bound to the same value. Hence it is sufficient to consider only descriptors with at most n inputs. Moreover, it was shown in previous work [9], that if the constraints $\mathcal{C}$ are weakly acyclic, n is upper-bounded by a polynomial in the size of Q whose largest exponent depends only on $\mathcal{C}$.

Therefore, the procedure **findDescriptors** can easily be extended to take parameters into account. The bottom-up step will infer descriptors in which the binding pattern component may contain up to n distinct input variables.

The following theorem summarizes the results of this section:

THEOREM 6.1. *If $\mathcal{C}$ is weakly acyclic, the following hold:*

- *Procedure* ***findDescriptors*** *extended with parameters outputs all pairwise dissimilar descriptors and is guaranteed to terminate in time exponential in the sizes of $\mathcal{P}$, $\mathcal{C}$ and Q.*
- *Procedure* ***testSupport*** *extended with parameters is a sound algorithm for checking support and runs in time exponential in the sizes of $\mathcal{P}$, $\mathcal{C}$ and Q. It becomes a complete decision procedure if $\mathcal{P}$ is a $\mathcal{C}$-local program generating $\mathcal{C}$-independent views.*

7. BOUNDARIES OF DECIDABILITY

In this section we explore the boundaries of decidability for the problems of expressibility and support. To calibrate our results, we start with the following: allowing unrestricted sets of constraints immediately leads to undecidability even if the program expresses a single view. This result is unsurprising given that unrestricted sets of embedded dependencies notoriously lead to undecidability of many fundamental database decision problems, such as equivalence of queries and implication of dependencies [1]:

THEOREM 7.1. *If $\mathcal{C}$ contains arbitrary embedded dependencies,* $\text{EXPR}^{\mathcal{C}}_{\mathcal{P}}(Q)$ *and* $\text{SUPP}^{\mathcal{C}}_{\mathcal{P}}(Q)$ *are undecidable even if $\mathcal{P}$ expresses a single view.*

PROOF. See Appendix A. □

Theorem 7.1 shows that decidability requires restrictions on the set of constraints, namely at least the restrictions leading to decidability for the single-view case. The least restrictive known condition for this purpose requires $\mathcal{C}$ to be a weakly acyclic set of embedded dependencies [9, 10]. As we show below, weak acyclicity turns out to be too generous for sets of views described by unrestricted programs, which justifies our key-safety restriction, showing that its relaxation leads to undecidability.

Indeed, it turns out that, when the program is not key-safe, the interaction of recursion in the program and the presence of dependencies leads to undecidability even under strong restrictions on the dependencies and on the program which are known to lead to decidability in many classical decision problems as long as recursion and dependencies are mutually exclusive. For instance, query rewritability using finitely many views (listed explicitly, not described by a program) is known to be decidable under weakly acyclic dependencies [9], in particular under only functional dependencies, or only full TGDs. In the absence of dependencies, expressibility and support for arbitrary (i.e. not necessarily key-safe) recursive programs is decidable [15, 26]. Moreover, many classical undecidable Datalog-related problems, such as containment and boundedness (undecidable by [12]) are known to become decidable for recursive monadic programs [6].

However when considering recursion and dependencies together, if we allow key-unsafe programs, we obtain surprisingly strong undecidability results, even for particular cases of weakly acyclic sets of constraints, such as a single key constraint (Theorem 7.2), and only full TGDs (Theorem 7.3).

THEOREM 7.2. *If $\mathcal{P}$ is recursive, then* $\text{EXPR}^{\mathcal{C}}_{\mathcal{P}}(Q)$ *is undecidable even if $\mathcal{C}$ consists of a single key constraint, and $\mathcal{P}$ is a linear monadic program.*

PROOF. See Appendix A. □

Theorems 7.2 and 3.2 immediately yield the undecidability of support under key constraints:

COROLLARY 7.1. *If $\mathcal{P}$ is recursive and not key-safe, then* $\text{SUPP}^{\mathcal{C}}_{\mathcal{P}}(Q)$ *is undecidable even if $\mathcal{C}$ consists of a single key constraint and $\mathcal{P}$ is a linear monadic program.*

THEOREM 7.3. *If $\mathcal{P}$ is recursive and not key-safe, then* $\text{EXPR}^{\mathcal{C}}_{\mathcal{P}}(Q)$ *is undecidable even if $\mathcal{C}$ contains only full TGDs and $\mathcal{P}$ is a monadic program.*

PROOF. See Appendix A. □

Theorems 7.3 and 3.2 yield undecidability of support under full TGDs:

COROLLARY 7.2. *If $\mathcal{P}$ is recursive and not key-safe, then* $\text{SUPP}^{\mathcal{C}}_{\mathcal{P}}(Q)$ *is undecidable even if $\mathcal{C}$ contains only full TGDs and $\mathcal{P}$ is a monadic program.*

Given that inclusion dependencies (INDs) are a particular case of TGDs, it is interesting to contrast Theorem 7.3 and Corollary 4.2. Notice that there is no contradiction here, as weakly acyclic sets of INDs and sets of full TGDs have incomparable expressive power. On one hand, weakly acyclic sets of INDs may include non-full TGDs. On the other, TGDs allow several subgoals in the premise whereas INDs allow only one.

8. RELATED WORK

The necessity of describing infinite families of views supported at the source was first argued in [20] and the problem of deciding support first solved (in the absence of dependencies) in [14, 15], which pioneers the idea of reducing the problem of support to that of rewriting a query using finitely many views. [15] compares the views generated by the program for *interchangeability*, meaning that V_1 and V_2 are interchangeable if in *every* rewriting R of Q, by replacing the V_1 goals with V_2 goals we still obtain a rewriting. [15] shows that interchangeability is an equivalence relation which induces finitely many equivalence classes and gives an algorithm which finds one representative of each class. We can show however that interchangeability under dependencies yields infinitely many equivalence classes, thus precluding the reduction from [15] (see Example B.1 in Appendix B). Even in the absence of dependencies, we observe that interchangeability is unnecessarily strong and the descriptor similarity condition (†) from Section 5 suffices. This allows us to manipulate mapping/partial mapping pairs (rather than *sets* thereof as done in [15]), which yields the improved upper bound.

[26] addresses both expressivity and support in the constraintless case. We improve on its upper bound for support. [26] claims for expressibility an EXPTIME lower bound even in the query size, which would make our EXPTIME upper bound tight. Unfortunately, the result is based on a reduction from containment of a CQ query in a Datalog program, claimed to be EXPTIME-hard in both query and program size, but which is actually in PTIME in the query size and thus does not transfer the EXPTIME lower bound to the query. The best lower bound we have remains NP.

The problem of support strictly extends that of rewriting queries using finitely many views. The latter was treated in depth in the literature, considering various extensions pertaining to the language of queries and views [13, 3, 2, 5], to adding limited access patterns for the views [11, 18], to adding constraints (see the references in [8]), and to mixing such extensions [7]. Prior work on information integration [16] studied answering queries using a finite set of views with limited access patterns with a different goal, that of finding maximally contained answers.

9. CONCLUSION

In this paper, we revisit the problem of deciding support and expressibility of a conjunctive query by views generated as the expansions of a Datalog program, investigating for the first time the effect of source constraints.

We identify practically relevant restrictions on the program which lead to decidability for weakly acyclic sets of key and foreign key constraints, which are the most prevalent constraints in practice. We present an algorithm which is applicable to unrestricted programs and any weakly acyclic set of embedded dependencies (which go beyond keys and foreign keys), yielding a decision procedure in all known decidable cases, and a sound test in general. Moreover, we show that our restrictions are maximally permissive, in the sense that their slightest relaxation leads to undecidability.

We also settle two problems left open by work on the constraint-free case, namely the exact complexity of deciding support and expressibility and the relationship between them.

First, we show that in the absence of constraints our algorithm is a decision procedure which improves the previously known upper bounds for support (from doubly-exponential time in [14] and non-deterministic exponential time in [26] to deterministic exponential in the size of both query and program). This algorithm has optimal running time w.r.t. the program size (we provide a matching EXPTIME lower bound for fixed query) and practically optimal running time w.r.t. the query size (we provide an NP lower bound for fixed program).

Second, we show that expressibility and support are inter-reducible in PTIME (even under constraints), which allows us to use essentially the same algorithm for solving them.

10. REFERENCES

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

[2] F. N. Afrati, R. Chirkova, M. Gergatsoulis, and V. Pavlaki. Finding equivalent rewritings in the presence of arithmetic comparisons. In *EDBT*, pages 942–960, 2006.

[3] F. N. Afrati, C. Li, and P. Mitra. Answering queries using views with arithmetic comparisons. In *PODS*, 2002.

[4] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *ACM Symposium on Theory of Computing (STOC)*, pages 77–90, 1977.

[5] S. Cohen, W. Nutt, and Y. Sagiv. Rewriting queries with arbitrary aggregation functions using views. *ACM Trans. Database Syst. (TODS)*, 31(2):672–715, 2006.

[6] S. Cosmadakis, H. Gaifman, P. Kanellakis, and M. Vardi. Decidable optimization problems for database logic programs. In *STOC*, pages 477–490, New York, NY, USA, 1988. ACM Press.

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

[8] A. Deutsch, L. Popa, and V. Tannen. Query reformulation with constraints. *SIGMOD Record*, 35(1):65–73, 2006.

[9] A. Deutsch and V. Tannen. Reformulation of XML queries and constraints. In *ICDT*, 2003.

[10] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: Semantics and query answering. In *ICDT*, 2003.

[11] D. Florescu, A. Y. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *SIGMOD*, pages 311–322, 1999.

[12] H. Gaifman, H. G. Mairson, Y. Sagiv, and M. Y. Vardi. Undecidable optimization problems for database logic programs. *Journal of the ACM (JACM)*, 40(3):683—713, 1993.

[13] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *PODS*, pages 95–104, 1995.

[14] A. Y. Levy, A. Rajaraman, and J. D. Ullman. Answering queries using limited external processors. In *PODS*, pages 227–237, 1996.

[15] A. Y. Levy, A. Rajaraman, and J. D. Ullman. Answering queries using limited external query processors. *J. Comput. Syst. Sci.*, 58(1):69–82, 1999.

[16] C. Li and E. Y. Chang. Query planning with limited source capabilities. In *ICDE*, pages 401–412, 2000.

[17] A. Motro. An access authorization model for relational databases based on algebraic manipulation of view definitions. In *ICDE*, pages 339–347. IEEE Computer Society, 1989.

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

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

[20] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. D. Ullman. A query translation scheme for rapid implementation of wrappers. In *DOOD*, pages 161–186, 1995.

[21] A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns. In *PODS*, pages 105–112. ACM Press, 1995.

[22] R. Ramakrishnan, Y. Sagiv, J. D. Ullman, and M. Y. Vardi. Proof-tree transformation theorems and their applications. In *PODS*, pages 172–181, 1989.

[23] S. Rizvi, A. O. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *SIGMOD*, pages 551–562, 2004.

[24] J. D. Ullman and J. E. Hopcroft. *Introduction to automata theory, languages and computation*. Addison-Wesley, 1979.

[25] V. Vassalos and Y. Papakonstantinou. Describing and using query capabilities of heterogeneous sources. In *VLDB*, pages 256–265, 1997.

[26] V. Vassalos and Y. Papakonstantinou. Expressive capabilities description languages and query rewriting algorithms. *J. Log. Program.*, 43(1):75–122, 2000.

APPENDIX

A. PROOFS

PROOF. (**Theorem 3.1**) We show that a conjunctive query Q is supported by a Datalog program $\mathcal{P}$ iff Q is expressible by a new program $\mathcal{P}'$ constructed from $\mathcal{P}$ w.r.t. Q.

The reduction starts from the following result, which generalizes a result of [13] to the presence of dependencies:

LEMMA A.1. *Let $\mathcal{C}$ be a weakly acyclic set of embedded dependencies. Then $\textsc{Supp}^{\mathcal{C}}_{\mathcal{P}}(Q)$ holds iff there is a rewriting R of Q under $\mathcal{C}$ using views generated by $\mathcal{P}$, where R has no more variables than $chase_{\mathcal{C}}(Q)$.*

It was shown in prior work [7] that, if $\mathcal{C}$ is weakly acyclic, then $chase_{\mathcal{C}}(Q)$ contains v variables, where v is upper-bounded by a polynomial in the number of goals in Q and exponential in the maximum arity of a relation appearing in the conclusion of a dependency in $\mathcal{C}$.

From this, we will build in PTIME in the size of $chase_{\mathcal{C}}(Q)$ and $\mathcal{P}$ a new program $\mathcal{P}'$ that basically enumerates all possible conjunctions of expansions of $\mathcal{P}$.

For this proof, it helps to consider Q and conjunctions of expansions as *rectified*. More precisely, no constants are allowed in predicate subgoals, and no variable appears twice in subgoals. Instead, joins are made explicit by subgoals $equals(X, Y)$, and selections with a constant c by subgoals $equals(X, c)$. Note that we can pass from any conjunctive query to its rectified version and vice-versa in linear time.

Given Q, denote with a_Q the arity of Q (the number of its distinguished variables). Assume w.l.o.g. that the distinguished predicate of $\mathcal{P}$ is *ans*, of arity $a_{\mathcal{P}}$. We add a new IDB predicate ans', as well as a new unary EDB predicate D.

We build the following program, of distinguished predicate ans':

$$\begin{array}{rcl}
ans'(V_1,\ldots,V_{a_Q}) & :- & pick(V_1,X_1,\ldots,X_v),\\
 & & pick(V_2,X_1,\ldots,X_v),\ldots\\
 & & pick(V_{a_Q},X_1,\ldots,X_v),\\
 & & temp(X_1,\ldots,X_v)\\
temp(X_1,\ldots,X_v) & :- & D(X_1),\ldots,D(X_v)\\
temp(X_1,\ldots,X_v) & :- & ans(Y_1,Y_2,\ldots,Y_{a_{\mathcal{P}}}),\\
 & & pick(Y_1,X_1,\ldots,X_v),\\
 & & pick(Y_2,X_1,\ldots,X_v),\ldots\\
 & & pick(Y_{a_{\mathcal{P}}},X_1,\ldots,X_v),\\
 & & temp(X_1,\ldots,X_v)\\
pick(V,X_1,\ldots,X_v) & :- & equals(V,X_1),\\
 & & D(V),D(X_1),\ldots,D(X_v)\\
pick(V,X_1,\ldots,X_v) & :- & equals(V,X_2),\\
 & & D(V),D(X_1),\ldots,D(X_v)\\
 & \vdots & \\
pick(V,X_1,\ldots,X_v) & :- & equals(V,X_v),\\
 & & D(V),D(X_1),\ldots,D(X_v)\\
+ & & \\
rules\ of\ \mathcal{P} & &
\end{array}$$

The rules added in addition to those of $\mathcal{P}$ have the task of expressing all possible conjunctive queries with a_Q head variables and at most v variables in total, formulated against the distinguished goal of $\mathcal{P}$, *ans*. The *ans* subgoals are then expanded into views generated by $\mathcal{P}$, due to the inclusion of the rules of $\mathcal{P}$ into $\mathcal{P}$'.

Note that the *temp* subgoal lists the pool of v variables the expansions of $\mathcal{P}'$ will use. Each *temp* subgoal expands into arbitrarily many *ans* subgoals which will build the body of the rewriting. The variables appearing in the head *ans'* and in the various *ans* subgoals in the body are each associated with *pick* subgoals. The *pick* subgoal has v possible expansions, each having the role of picking one of the v variables in the pool to equate with the variable in its first argument. In this way, every assignment of variables from the pool to variables of the head and body of the conjunctive query over *ans* subgoals is realizable by some expansion of the *pick* subgoals.

The D predicate is introduced for technical purposes, to avoid generating unsafe Datalog rules for the *pick* goal. Its effect is that each view generated by $\mathcal{P}$' has a D subgoal for each of its variables. This does not influence expressibility as long as we add such subgoals for all variables appearing in the query and in the dependencies. Indeed, if Q has the form

$$Q(Z_1,\ldots,Z_{a_Q}) \quad :- \quad body(Z_1,\ldots,Z_{v_Q})$$

with *body* a conjunction of subgoals, we build a new boolean query

$$\begin{array}{rcl}
Q'(Z_1,\ldots,Z_{a_Q}) & :- & body(Z_1,\ldots,Z_{v_Q}),\\
 & & D(Z_1),\ldots,D(Z_{v_Q})
\end{array}$$

Finally, we construct a new set of dependencies $\mathcal{C}$' by adding in each dependency σ from $\mathcal{C}$ the predicate $D(X)$ for every variable X appearing in σ.

Notice that $\mathcal{C}'$ and Q' are obtained in linear time from $\mathcal{C}$ and Q, respectively. $\mathcal{P}'$ is obtained in PTIME from $\mathcal{P}$ and v, where the latter is polynomial in the size of Q but exponential in the maximum arity of a relation appearing in the conclusion of some dependency from $\mathcal{C}$.

It is easy to show that $\textsc{Supp}^{\mathcal{C}}_{\mathcal{P}}(Q)$ holds if and only if
$\textsc{Expr}^{\mathcal{C}'}_{\mathcal{P}'}(Q')$ does. □

PROOF. (**Theorem 3.2**) Given Q,$\mathcal{P}$ and $\mathcal{C}$, we construct a boolean query Q'', boolean program $\mathcal{P}''$ and set of dependencies $\mathcal{C}''$, such that Q is expressible by $\mathcal{P}$ under $\mathcal{C}$ iff Q'' is supported by $\mathcal{P}''$ under $\mathcal{C}''$.

For presentation simplicity, we first show a first-cut solution which works only if the query graph is connected, then we explain how the reduction can be adapted to arbitrary queries.

Denote with a_Q the arity of Q and assume w.l.o.g. that Q has the form

$$Q(Z_1,\ldots,Z_{a_Q}) \quad :- \quad body(Z_1,\ldots,Z_{v_Q})$$

with *body* a conjunction of subgoals and $v_Q \geq a_Q$ the total number of variables appearing in Q. We build the boolean query

$$Q'() \quad :- \quad head(Z_1,\ldots,Z_{a_Q}), body(Z_1,\ldots,Z_{v_Q})$$

using a fresh EDB relation *head* of arity a_Q.

Assume w.l.o.g. that the distinguished IDB of $\mathcal{P}$ is *ans*. Notice that, for Q to be expressible by $\mathcal{P}$, *ans* must have the same arity as Q. $\mathcal{P}$' is constructed by adding to the rules of $\mathcal{P}$ a new rule defining a fresh, boolean IDB predicate *ans'*:

$$ans'() \quad :- \quad ans(X_1,\ldots,X_{a_Q}), head(X_1,\ldots,X_{a_Q}).$$

The distinguished IDB predicate of $\mathcal{P}$' is *ans'*.

Note that the views generated by $\mathcal{P}$' are in one-to-one correspondence to those generated by $\mathcal{P}$: any view V' generated by $\mathcal{P}$' simply extends the body of some view V generated by $\mathcal{P}$ with a *head* subgoal containing the head variables of V. Q is equivalent to V if and only if Q' is equivalent to the corresponding view V': the *head* subgoals appearing in both Q' and V' ensure the desired correspondence between the distinguished variables of Q and those of V. We have thus proven

Claim 1. $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}'}(Q')$ iff $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}}(Q)$.

Also note that, since each view generated by $\mathcal{P}$' is boolean, any rewriting using such views is really a Cartesian product thereof. We therefore make the following claim:

Claim 2. Consider a boolean query Q'' and the set of embedded dependencies $\mathcal{C}''$. If

(a) Q'' performs no Cartesian products (i.e. if its hypergraph [1] is connected), and
(b) all constraints in $\mathcal{C}''$ have premises with connected hypergraph,

then Q'' is equivalent under $\mathcal{C}''$ to some boolean conjunctive query R iff it is equivalent under $\mathcal{C}''$ to a connected subquery of R. $\diamond$

Proof of Claim 2. The "if" direction is immediate, we prove the "only if" direction next.

Let Q'' be connected, and $R() :- \ V_1(), V_2()$, where the hypergraphs of V_1 and V_2 are disjoint.

Assume toward a contradiction that $V_1 \not\subseteq_{\mathcal{C}''} V_2$ and $V_2 \not\subseteq_{\mathcal{C}''} V_1$. Then there must exist two databases, DB_1, DB_2, with disjoint active domains, such that both DB_1, DB_2 satisfy $\mathcal{C}''$, and such that $V_1(DB_1) = true$, $V_2(DB_1) = false$, $V_1(DB_2) = false$ and $V_2(DB_2) = true$. Since Q'' is equivalent to R under $\mathcal{C}''$, we obtain that $Q(DB_1) = Q(DB_2) = false$.

Let DB_3 be the database obtained by unioning the two:
$DB_3 := DB_1 \cup DB_2$.

We claim that DB_3 satisfies $\mathcal{C}''$ as well: the components DB_1, DB_2 do so by hypothesis, and their disjoint union cannot violate any constraint in $\mathcal{C}''$ because all constraint premises are connected and thus cannot match across the databases.

Note that $Q''(DB_3) = false$, as Q'' has no match into any of DB_1, DB_2, and no match across them because it is connected. Also note that $R(DB_3) = true$, as V_1 and V_2 have a match against the sub-databases DB_1 and DB_2, respectively.

We have thus exhibited a database $DB_3 \models \mathcal{C}''$ such that $R(DB_3) = true$, but $Q''(DB_3) = false$, contradicting the equivalence of Q to R under $\mathcal{C}''$. Therefore, either of V_1, V_2 must be contained in the other under $\mathcal{C}''$, so R can be minimized under $\mathcal{C}''$ to just one component. The reasoning extends to arbitrarily many components by induction.
End of proof of Claim 2.

By Claim 2, we have that, under restrictions (a) and (b), all rewritings of Q' under $\mathcal{C}$ using views generated by $\mathcal{P}$' contain a single view goal or can be minimized to a single view goal. This implies that $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}'}(Q')$ holds if and only if $\textsc{Supp}^{\mathcal{C}}_{\mathcal{P}'}(Q')$ does. Considering also Claim 1, we obtain $\textsc{Supp}^{\mathcal{C}}_{\mathcal{P}'}(Q')$ iff $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}'}(Q')$ iff $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}}(Q)$.

We now refine the reduction, lifting restrictions (a) and (b). To this end, we obtain from $Q', \mathcal{P}'$ and $\mathcal{C}$, $Q'', \mathcal{P}''$ and $\mathcal{C}$" such that $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}}(Q)$ holds iff $\textsc{Expr}^{\mathcal{C}''}_{\mathcal{P}''}(Q'')$ does, and such that Q'' and the premises of all constraints in $\mathcal{C}''$ are connected. Then Claim 2 will apply to Q'' and $\mathcal{C}''$, completing the proof.

The head of Q'' is the same as that of Q'. The distinguished IDB of $\mathcal{P}''$ is the same as that of $\mathcal{P}$'. Every remaining goal and subgoal of Q' and $\mathcal{P}'$, say $G(\bar{X})$ of arity a, is extended to an $a+1$-ary goal $G(\bar{X}, U)$, where U is a fresh variable shared across all goals.

We replace in the same way all subgoals appearing in dependencies in $\mathcal{C}$: for every $\sigma \in \mathcal{C}$ of form

$$\forall \bar{X}\ premise(\bar{X}) \rightarrow \exists \bar{Y}\ conclusion(\bar{X}, \bar{Y}),$$

we construct σ'' of form

$$\forall \bar{X} \forall U\ premise''(\bar{X}, U) \rightarrow \exists \bar{Y}\ conclusion''(\bar{X}, \bar{Y}, U),$$

where *premise"* and *conclusion"* are obtained from *premise* and *conclusion*, respectively, by extending the goals with the new variable U, as done above for Q' and $\mathcal{P}'$.

Claim 3. $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}'}(Q')$ holds iff $\textsc{Expr}^{\mathcal{C}''}_{\mathcal{P}''}(Q'')$ does.

The theorem follows from Claims 1, 3 and 2. □

Proof. (**Theorem 7.1**) The undecidability of support follows from the undecidability of expressibility and the reduction of Theorem 3.2. As for the undecidability of expressibility, it follows from a reduction from query containment under embedded dependencies (known to be undecidable [1]) to support of a query by a non-recursive Datalog program which expresses a single view. □

Proof. (**Theorem 7.2**) The proof is by reduction from the Post Correspondence Problem (PCP), known to be undecidable [24, 1]. Let $\{v_i\}_{1 \leq i \leq n}$, $\{w_i\}_{1 \leq i \leq n}$ be the PCP instance, where v_i, w_i are words over alphabet $\{a, b\}$. This is a "yes" instance iff there exists a natural number l and a sequence of integers $\sigma \in \{1, \ldots, n\}^l$ such that

$$v_{\sigma(1)} \circ v_{\sigma(2)} \circ \ldots \circ v_{\sigma(l)} = w_{\sigma(1)} \circ w_{\sigma(2)} \circ \ldots \circ w_{\sigma(l)}$$

where $\sigma(i)$ denotes the i^{th} integer in the sequence, and $\circ$ is the word concatenation operator. Any such σ is called a *solution* of the PCP problem. Any sequence σ (regardless of whether it is a solution) determines a word obtained by concatenating the corresponding w-words, and one obtained by concatenating the corresponding v-words.

We construct a monadic, linear (recursive) Datalog program $\mathcal{P}$, the singleton set $\mathcal{C}$ comprising a key constraint, and a query Q such that the PCP problem has a solution iff $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}}(Q)$.

We use only one EDB relation $e(X, l, Y)$, intended to denote a directed edge with source X, target Y and label l. The (boolean) query Q is the following, where all lower-case letters (e.g. l, r, a, b, c, d) are constants, and upper-case letters are variables:

$$\begin{array}{lll} Q() & :- & e(A, l, B), e(A, r, C), e(D, c, A), \\ & & e(D, a, D), e(D, b, D), e(D, d, D). \end{array}$$

The program $\mathcal{P}$ is constructed as follows (again, lower-case letters are constants and upper-case letters are variables). $\mathcal{P}$ consists of

- the rule $V() :- \ C(X)$;
- the rule

$$\begin{array}{lll} C_r(X) & :- & e(X, d, Y), \\ & & e(X, c, X'), e(X', l, Z), \\ & & e(Y, c, Y'), e(Y', r, T), \\ & & e(U, a, U), e(U, b, U), e(U, d, U), \\ & & e(U, c, X'); \end{array}$$

- for every $1 \leq i \leq n$, assuming w.l.o.g. that $v_i = \alpha_1^i \dots \alpha_{k_i}^i$ and $w_i = \beta_1^i \dots \beta_{l_i}^i$, the rules

$$\begin{array}{lll} C(X) & :- & e(X, \alpha_1^i, X_1), \dots, e(X_{k_i - 1}, \alpha_{k_i}^i, X_{k_i}), \\ & & e(X, \beta_1^i, Y_1), \dots, e(Y_{l_i - 1}, \beta_{l_i}^i, Y_{l_i}), \\ & & e(X_{k_i}, d, Y_{l_i}), C_r(X_{k_i}); \\ C_r(X) & :- & e(X, d, Y), \\ & & e(X, \alpha_1^i, X_1), \dots, e(X_{k_i - 1}, \alpha_{k_i}^i, X_{k_i}), \\ & & e(Y, \beta_1^i, Y_1), \dots, e(Y_{l_i - 1}, \beta_{l_i}^i, Y_{l_i}), \\ & & e(X_{k_i}, d, Y_{l_i}), C_r(X_{k_i}); \end{array}$$

$\mathcal{C}$ comprises just one key constraint, stating that the source and label of an edge determine its target:

$$\forall X, L, Y, Y' \; e(X, L, Y) \wedge e(X, L, Y') \rightarrow Y = Y'.$$

$\mathcal{P}$ is designed to generate, for every sequence σ of integers from $\{1, \dots, n\}$, an expansion which encodes the two concatenations of v-words and w-words determined by σ. A word is encoded by a chain of edges, each edge label encoding a character in the word. The expansion thus contains two chains of words (one for the v_i's, one for the w_i's), each of them ended by a c-labeled edge followed by an l-edge, respectively an r-edge. The chains start from the same node (according to the C rule), and continue in parallel, chaining together pairs of subchains which correspond to pairs of words (v_i, w_i) for some i (this is the role of the repeated expansions of IDB C_r according to the rule for i). The expansion is ended by a subgraph given by the expansion of the first rule of C_r, whose role will be explained shortly.

To enable mappings from the arbitrarily long chains of the expansions into the query, Q contains cycles into which every pair of chains can map. Indeed, it is easy to see that any expansion of $\mathcal{P}$ has a containment mapping into Q. Since the cycles in Q cannot map into the straight chains in the expansions of $\mathcal{P}$, the v-chain is ended by the cycles generated by the first rule of C_r.

We therefore have that $\textsc{Expr}_{\mathcal{P}}^{\mathcal{C}}(Q)$ holds iff $\mathcal{P}$ expresses some view V such that $V \sqsubseteq_{\mathcal{C}} Q$ (since the opposite containment holds for every expansion, even in the absence of constraints). Because $\mathcal{C}$ contains only a key constraint, the chase with it is guaranteed to terminate, and $V \sqsubseteq_{\mathcal{C}} Q$ holds iff $\mathit{chase}_{\mathcal{C}}(V) \sqsubseteq Q$ [1].

Observe that successive expansions of the C_r IDB chain only the v-words together; the v_i-words in the expansion of each rule start from variable X which is also the end of the previous v_j-word in the concatenation, but the w_i-words start from the fresh variable Y which is not connected to the end Y_{k_j} of the previous w_j-word. Connecting the successive w-words explicitly would require IDB C_r to be binary, carrying both ends of v and w-words. To use only monadic rules, we rely instead on the key constraint: the variable beginning any w-word and the variable ending the previous w-word in the chain are both targets of d-edges emanating from the junction of the previous and current v-word. The chase with the key constraint will "glue" the two chain segments corresponding to the w-words.

The intuition behind the construction is that, if we log for each one-step expansion of IDB C_r the i corresponding to the rule used, the obtained sequence of integers is the candidate for the solution of the PCP problem. All possible sequences of one-step expansions thus generate all possible solution candidates.

The theorem follows from the following claim, stating that a candidate solution is verified as a true solution only by finding a containment mapping from Q into the chase result of the corresponding expansion:

Claim. There is some view V generated by $\mathcal{P}$ such that

$$\mathit{chase}_{\mathcal{C}}(V) \sqsubseteq Q$$

iff V encodes a solution of the PCP problem. $\diamond$

Proof. Notice that the chase of any expansion E with the key constraint can only start at the common origin of the v- and w-chains, and can only continue as long as the labels in the chains situated at the same distance from the origin coincide. The chains are determined by a solution to the PCP problem if and only if they match on their entire length, which is equivalent to the chase proceeding to collapse the chains all the way to their ends. This is detected by the fact that the l- and the r-edges eventually share the same source, which in turn is the only way in which the query pattern can map into the chase result of E.
End of proof of claim.

□

PROOF. (**Theorem 7.3**) We use a reduction from PCP, adapting the construction from the proof of Theorem 7.2. The main difficulty here is to control that the two chains of v- and w-words are determined by the same sequence of integers, and that the chains match each other in length and labels. This was achieved in the proof of Theorem 7.2 by chasing with the key constraint.

We introduce fresh edge labels, $i_1, \dots, i_n$, for n being the number of PCP words. We also use the labels $sync$, end, up, $down$.

We construct a monadic, (recursive) Datalog program $\mathcal{P}$, the set $\mathcal{C}$ comprising three families of TGDs, and a query Q such that the PCP problem has a solution iff $\textsc{Expr}_{\mathcal{P}}^{\mathcal{C}}(Q)$.

The Datalog program $\mathcal{P}$ contains:

- a rule for the distinguished IDB predicate *ans*: $ans() :- C(X)$;
- for every $1 \leq i \leq n$, assuming w.l.o.g. that $v_i = \alpha_1^i \dots \alpha_{k_i}^i$ and $w_i = \beta_1^i \dots \beta_{l_i}^i$, the rules

$$\begin{array}{lll} C(X) & :- & e(X, sync, X), \\ & & e(X, \alpha_1^i, X_1), \dots, e(X_{k_i - 1}, \alpha_{k_i}^i, X_{k_i}), \\ & & e(X, \beta_1^i, Y_1), \dots, e(Y_{l_i - 1}, \beta_{l_i}^i, Y_{l_i}), \\ & & e(X, i, X_{k_i}), e(X, i, Y_{l_i}), \\ & & C_v(X_{k_i}), C_w(Y_{l_i}); \\ & & \\ C_v(X) & :- & e(X, \alpha_1^i, X_1), \dots, e(X_{k_i - 1}, \alpha_{k_i}^i, X_{k_i}), \\ & & e(X, i, X_{k_i}), C_v(X_{k_i}); \\ & & \\ C_w(X) & :- & e(X, \beta_1^i, Y_1), \dots, e(Y'_{l_i - 1}, \beta_{l_i}^i, Y_{l_i}), \\ & & e(X, i, Y_{l_i}), C_w(Y_{l_i}); \end{array}$$

- the rules

$$\begin{array}{lll} C_v(X) & :- & e(X, end, Y), e(Y, up, Z) \\ C_w(X) & :- & e(X, end, Y), e(Y, down, Z) \\ & & e(X, a, X), e(X, b, X), \\ & & e(X, i_1, X), \dots, e(X, i_n, X), \\ & & e(X, sync, X) \end{array}$$

The program expands into chains that are not necessarily synchronized. We control synchronization by constraints. More precisely, we use TGDs to control that the two chains are determined by the same sequence of integers, and to control that the two chains match. $\mathcal{C}$ comprises:

- for each $1 \leq i \leq n$, the full TGD

$$\forall X, Y, Z, T \tag{4}$$
$$e(X, sync, Y) \wedge e(X, i, Z) \wedge (Y, i, T) \rightarrow e(Z, sync, T)$$

- for each $l, l' \in \{a, b\}$, the full TGD

$$\forall X, Y, Z, T \tag{5}$$
$$e(X, l, Y), e(X, l, Z), e(Y, l', T) \rightarrow e(Z, l', T)$$

- for each $l \in \{a, b\}$, the full TGD

$$\begin{aligned} &\forall X, Y, Z, T, U, V, W \qquad (6)\\ &e(X, l, Y), e(X, l, Z), e(Z, end, T),\\ &e(Y, end, V), e(T, up, U), e(V, down, W),\\ &e(Y, sync, Z) \rightarrow e(V, up, U) \end{aligned}$$

Intuitively, an expansion has an end-edge to signal the end of each chain, then an up-edge to signal the end of the chain of v-words, and a $down$-edge to signal the end of the chain of w-words.

The $sync$ edges are added by the chase of the expansion with the family of TGDs (4), to mark the pairs of nodes on the two chains which represent chain prefixes determined by the same sequence of integers from $\{1, \ldots, n\}$.

Since the two chains of the expansion have a common origin (due to the expansion of IDB C), the chase with the family of TGDs (5) can only start at the origin, and continues down the chains only as far as the labels of the chain prefixes match. The two chains match entirely if and only if the chase with (5) stops at the chain ends (marked by end-edges).

If the chase with both families of TGDs (4) and (5) goes all the way to the end of the two chains, then both the sequence of integers and the sequence of labels coincide, hence the chains encode a PCP solution. This is detected by the family of TGDs (6), which apply only in that case, recording this fact by copying the up-edge from the end of the v-chain to the end of the w-chain, thus creating a node with both up and $down$ edges emanating from it.

This is precisely what the query checks for:

$$\begin{aligned} Q : q() \quad :- \quad & e(T, a, T), e(T, b, T)\\ & e(T, i_1, T), \ldots, e(T, i_n, T),\\ & e(T, sync, T),\\ & e(T, end, X), e(X, up, Y), e(X, down, Z) \end{aligned}$$

Similar to proof of Theorem 7.2, in order to enable mappings from the arbitrarily long chains of the expansions into the query, Q contains cycles into which every pair of chains can map. Indeed, it is easy to see that any expansion of $\mathcal{P}$ has a containment mapping into Q. Since the cycles in Q cannot map into the straight chains in the expansions of $\mathcal{P}$, the w-chain is ended by the cycles generated by the second rule of C_w.

It is easy to verify that Q can be mapped into the result of chasing some expansion of $\mathcal{P}$ with $\mathcal{C}$ iff the expansion encodes a PCP solution. □

PROOF. (**Lemma 5.2**) (a) Notice that the initialization stage and each individual rule step terminate, since the chase terminates when $\mathcal{C}$ is weakly acyclic. The set $\mathcal{D}$ must saturate, as there are only finitely many dissimilar descriptors. Their number is upper bounded by an exponential in the maximum arity of a predicate in $\mathcal{P}$ and the size of Q, which bounds the number of rule step applications. At every rule step, finding that the rule applies involves matching it against the set of descriptors, which is exponential in the rule size. By Theorem C.1, the ensuing chase terminates in time exponential in the size of $\mathcal{C}$ and polynomial in the size of the descriptor.

(b) An easy proof by induction on the structure of the derivation tree of each descriptor. □

PROOF. (**Theorem 5.3**) The NP lower bound follows from a reduction from the problem of checking conjunctive query equivalence (NP-complete by [4]), via the problem of checking expressibility. Given conjunctive queries Q_1, Q_2, we have that $Q_1 \equiv Q_2$ iff Q_1 is expressible by the single-rule Datalog program Q_2. The latter reduces in PTIME to the problem of support by Theorem 3.2.

The EXPTIME lower bound is obtained by a reduction from the problem of checking containment of a query Q in a Datalog program $\mathcal{P}$, known to be PTIME in the size of Q and EXPTIME-complete in the size of $\mathcal{P}$ [22]. First, we carry out a reduction to the problem of checking expressibility, then compose it with the PTIME reduction from expressibility to support given by Theorem 3.2:

Given query $Q(\bar{X}) \ :- \ body(\bar{X}, \bar{Y})$ and program $\mathcal{P}$ of distinguished predicate ans (necessarily of same arity as the query), we construct program $\mathcal{P}'$ which includes all rules of $\mathcal{P}$, the additional rule $ans'(\bar{Z}) \ :- \ ans(\bar{Z}), body(\bar{Z}, \bar{Y})$ and pick ans' as the new distinguished predicate of $\mathcal{P}'$. Notice that $\mathcal{P}'$ generates all intersections of Q with views generated by $\mathcal{P}$, whence we have that Q is contained in $\mathcal{P}$ iff $\text{EXPR}^{\emptyset}_{\mathcal{P}'}(Q)$ holds. □

B. INTERCHANGEABILITY DOES NOT HELP

The following example shows that under dependencies, there are infinitely many equivalence classes of views with respect to interchangeability. This precludes the reduction described in [15] from the problem of support to that of rewriting using finitely many views, as it involves focusing on representatives of the equivalence classes.

EXAMPLE B.1. *We have a program $\mathcal{P}$ that produces unary views as follows:*

$$\begin{aligned} V(X) \quad &:- \quad e(X, a, Y), C_r(Y)\\ C_r(X) \quad &:- \quad e(X, a, Y), C_r(Y)\\ C_r(X) \quad &:- \quad e(X, b, Y), e(Y', b, Y), e(Y', a, Y'),\\ & \qquad e(Y, up, Z)\\ C_r(X) \quad &:- \quad e(X, b, Y), e(Y', b, Y), e(Y', a, Y'),\\ & \qquad e(Y, down, Z) \end{aligned}$$

Expansions are chains of a-labeled edges ending with a b-labeled edge and one of up *or* down.

Consider the query Q:

$$\begin{aligned} Q() \quad :- \quad & e(D, a, D), e(D, b, A),\\ & e(A, up, B), e(A, down, C) \end{aligned}$$

The source obeys also one key constraint for each $l \in \{a, b\}$:

$$\forall X, Y', Y'' \; e(X, l, Y'), e(X, l, Y'') \longrightarrow Y' = Y''.$$

We write V_n for the expansion with n a-labeled edges and ending with up. *We write U_n for the expansion with n a-labeled edges and ending with* down.

We can see that, for any n, the rewriting R_n defined as $R_n() : -V_n(X), U_n(X)$ is an equivalent rewriting of Q.

However, replacing in R_n the V_n goal with any other view (V_i or U_i) would not yield another equivalent rewriting. So each V_n

(and each U_n) is in its own equivalence class w.r.t. interchangeability in rewritings for Q. There are therefore infinitely many such equivalence classes.

C. WEAK ACYCLICITY

We repeat for the reader's convenience the definition of weakly acyclic set of dependencies, and the associated result.

DEFINITION C.1. ***(Weakly Acyclic)****[9, 10] A* position *is a pair (R, i) (which we write R^i) where R is a relation symbol of arity r and i satisfies $1 \leq i \leq r$. The* dependency graph *of a set Σ of TGDs is a directed graph where the vertices are the positions of the relation symbols in Σ and, for every TGD ξ of the form*

$$\forall \bar{u}, \bar{w}\; \phi(\bar{u}, \bar{w}) \longrightarrow \exists \bar{v}\; \psi(\bar{u}, \bar{v})$$

there is an edge between R^i and S^j whenever (1) some $u \in \{\bar{u}\}$ occurs in R^i in ϕ and in S^j in ψ or (2) some $u \in \{\bar{u}\}$ appears in R^i in ϕ and some $v \in \{\bar{v}\}$ occurs in S^j in ψ. Furthermore, these latter edges are labeled with $\exists$ and we call them existential edges. *A set Σ of TGDs and EGDs is* weakly acyclic *if the dependency graph of its TGD set has no cycles through an existential edge.*

THEOREM C.1. *For every weakly acyclic set $\mathcal{C}$ of embedded dependencies, there are b and c such that, for any set of subgoals A, regardless of the order of the chase, $chase_{\mathcal{C}}(A)$ is guaranteed to terminate in $O(|A|^b)$ steps and in time $O(|A|^c)$, where $|A|$ denotes the size of A.*