Equivalent Rewritings on Path Views with Binding Patterns

Julien Romero, Nicoleta Preda, Antoine Amarilli, Fabian M. Suchanek

Equivalent Rewritings on Path Views with Binding Patterns (Extended Version)

Julien Romero[1], Nicoleta Preda[2], Antoine Amarilli[1], and Fabian Suchanek[1]

[1] LTCI, Télécom Paris, Institut Polytechnique de Paris, France
first.last@telecom-paris.fr
[2] Université de Versailles nicoleta.preda@uvsq.fr

Abstract. A view with a binding pattern is a parameterized query on a database. Such views are used, e.g., to model Web services. To answer a query on such views, the views have to be orchestrated together in execution plans. We show how queries can be rewritten into equivalent execution plans, which are guaranteed to deliver the same results as the query on all databases. We provide a correct and complete algorithm to find these plans for path views and atomic queries. Finally, we show that our method can be used to answer queries on real-world Web services.

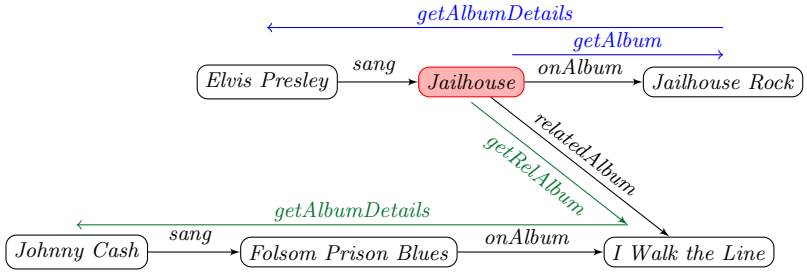

Fig. 1: An equivalent execution plan (blue) and a maximal contained rewriting (green) executed on a database (black).

1 Introduction

In this paper, we study views with binding patterns [26]. Intuitively, these can be seen as functions that, given input values, return output values from a database. For example, a function on a music database could take as input a musician, and return the songs by the musician stored in the database.

Several databases on the Web can be accessed only through such functions. They are usually presented as a form or as a Web service. For a REST Web service, a client calls a function by accessing a parameterized URL, and it responds by sending back the results in an XML or JSON file. The advantage of such an

interface is that it offers a simple way of accessing the data without downloading it. Furthermore, the functions allow the data provider to choose which data to expose, and under which conditions. For example, the data provider can allow only queries about a given entity, or limit the number of calls per minute. According to programmableWeb.com, there are over 20,000 Web services of this form – including LibraryThing, Amazon, TMDb, Musicbrainz, and Lastfm.

If we want to answer a user query on a database with such functions, we have to *compose* them. For example, consider a database about music – as shown in Figure 1 in black. Assume that the user wants to find the musician of the song *Jailhouse*. One way to answer this query is to call a function *getAlbum*, which returns the album of the song. Then we can call *getAlbumDetails*, which takes as input the album, and returns all songs on the album and their musicians. If we consider among these results only those with the song *Jailhouse*, we obtain the musician *Elvis Presley* (Figure 1, top, in blue). We will later see that, under certain conditions, this plan is guaranteed to return exactly all answers to the query on all databases: it is an *equivalent rewriting* of the query. This plan is in contrast to other possible plans, such as calling *getRelatedAlbum* and *getAlbumDetails* (Figure 1, bottom, in green). This plan does not return the exact set of query results. It is a *maximally contained rewriting*, another form of rewriting, which we will discuss in the related work.

Equivalent rewritings are of primordial interest to the user because they allow obtaining exactly the answers to the query – no matter what the database contains. Equivalent rewritings are also of interest to the data provider: For example, in the interest of usability, the provider may want to make sure that equivalent plans can answer all queries of importance. However, finding equivalent rewritings is inherently non-trivial. As observed in [2, 4], the problem is undecidable in general. Indeed, plans can recursively call the same function. Thus, there is, a priori, no bound on the length of an execution plan. Hence, if there is no plan, an algorithm may try forever to find one – which indeed happens in practice.

In this paper, we focus on path functions (i.e., functions that form a sequence of relations) and atomic queries. For this scenario, we can give a correct and complete algorithm that decides in PTIME whether a query has an equivalent rewriting or not. If it has one, we can give a grammar that enumerates all of them. Finally, we show that our method can be used to answer queries on real-world Web services. After reviewing related work in Section 2 and preliminaries in Section 3, we present our problem statement in Section 4 and our algorithm in Section 5, concluding with experiments in Section 6. This is an extended version of the conference paper which contains all detailed proofs in the appendix.

2 Related Work

Formally, we aim at computing *equivalent rewritings* over views with binding patterns [26] in the presence of inclusion dependencies. Our approach relates to the following other works.

Equivalent Rewritings. Checking if a query is *determined* by views [16], or finding possible equivalent rewritings of a query over views, is a task that has been intensively studied for query optimization [4, 15], under various classes of constraints. In our work, we are specifically interested in computing equivalent rewritings over views with binding patterns, i.e., restrictions on how the views can be accessed. This question has also been studied, in particular with the approach by Benedikt et al. [2] based on logical interpolation, for very general classes of constraints. In our setting, we focus on path views and unary inclusion dependencies on binary relations. This restricted (but practically relevant) language of functions and constraints has not been investigated in [2]. We show that, in this context, the problem is solvable in PTIME. What is more, we provide a self-contained, effective algorithm for computing plans, for which we provide an implementation. We compare experimentally against the PDQ implementation by Benedikt et al. [3] in Section 6.

Maximally Contained Rewritings. Another line of work has studied how to rewrite queries against data sources in a way that is not equivalent but maximizes the number of query answers [17]. Unlike equivalent rewritings, there is no guarantee that all answers are returned. For views with binding patterns, a first solution was proposed in [13, 14]. The problem has also been studied for different query languages or under various constraints [7, 8, 12, 21]. We remark that by definition, the approach requires the generation of relevant but not-so-smart call compositions. These call compositions make sure that no answers are lost. Earlier work by some of the present authors proposed to prioritize promising function calls [22] or to complete the set of functions with new functions [23]. In our case, however, we are concerned with identifying only those function compositions that are guaranteed to deliver answers.

Orthogonal Works. Several works study how to optimize given execution plans [29, 32]. Our work, in contrast, aims at *finding* such execution plans. Other works are concerned with mapping several functions onto the same schema [10, 19, 31]. Our approach takes a Local As View perspective, in which all functions are already formulated in the same schema.

Federated Databases. Some works [25, 28] have studied *federated databases*, where each source can be queried with any query from a predefined language. By contrast, our sources only publish a set of preset parameterized queries, and the abstraction for a Web service is a view with a binding pattern, hence, a predefined query with input parameters. Therefore, our setting is different from theirs, as we cannot send arbitrary queries to the data sources: we can only call these predefined functions.

Web Services. There are different types of Web services, and many of them are not (or cannot be) modeled as views with binding patterns. AJAX Web services use JavaScript to allow a Web page to contact the server. Other Web services are used to execute complex business processes [11] according to protocols or choreographies, often described in BPEL [30]. The Web Services Description Language (WSDL) describes SOAP Web services. The Web Services Modeling Ontology (WSMO) [33], in the Web Ontology Language for Services (OWL-

S) [20], or in Description Logics (DL) [27] can describe more complex services. These descriptions allow for Artificial Intelligence reasoning about Web services in terms of their behavior by explicitly declaring their preconditions and effects. Some works derive or enrich such descriptions automatically [6,9,24] in order to facilitate Web service discovery.

In our work, we only study Web services that are querying interfaces to databases. These can be modeled as views with binding patterns and are typically implemented in the Representational State Transfer (REST) architecture, which does not provide a formal or semantic description of the functions.

3 Preliminaries

Global Schema. We assume a set $\mathcal{C}$ of constants and a set $\mathcal{R}$ of relation names. We assume that all relations are binary, i.e., any n-ary relations have been encoded as binary relations by introducing additional constants[3]. A *fact* $r(a, b)$ is formed using a relation name $r \in \mathcal{R}$ and two constants $a, b \in \mathcal{C}$. A *database instance* I, or simply *instance*, is a set of facts. For $r \in \mathcal{R}$, we will use r^- as a relation name to mean the inverse of r, i.e., $r^-(b, a)$ stands for $r(a, b)$. More precisely, we see the inverse relations r^- for $r \in \mathcal{R}$ as being relation names in $\mathcal{R}$, and we assume that, for any instance I, the facts of I involving the relation name r^- are always precisely the facts $r^-(b, a)$ such that $r(a, b)$ is in I.
Inclusion Dependencies. A *unary inclusion dependency* for two relations r, s, which we write $r \rightsquigarrow s$, is the following constraint:

$$\forall x, y : r(x, y) \Rightarrow \exists z : s(x, z)$$

Note that one of the two relations or both may be inverses. In the following, we will assume a fixed set $\mathcal{UID}$ of unary inclusion dependencies, and we will only consider instances that satisfy these inclusion dependencies. We assume that $\mathcal{UID}$ is closed under implication, i.e., if $r \rightsquigarrow s$ and $s \rightsquigarrow t$ are two inclusion dependencies in $\mathcal{UID}$, then so is $r \rightsquigarrow t$.
Queries. An *atom* $r(\alpha, \beta)$ is formed with a relation name $r \in \mathcal{R}$ and α and β being either constants or variables. A *query* takes the form

$$q(\alpha_1, ..., \alpha_m) \leftarrow B_1, ..., B_n$$

where $\alpha_1, ...\alpha_m$ are variables, each of which must appear in at least one of the body atoms $B_1, ...B_n$. We assume that queries are *connected*, i.e., each body atom must be transitively linked to every other body atom by shared variables. An *embedding* for a query q on a database instance I is a substitution σ for the variables of the body atoms so that $\forall B \in \{B_1, ..., B_n\} : \sigma(B) \in I$. A *result* of a query is an embedding projected to the variables of the head atom. We write $q(\alpha_1, ..., \alpha_m)(I)$ for the results of the query on I. An *atomic query* is a query that takes the form $q(x) \leftarrow r(a, x)$, where a is a constant and x is a variable.

[3] https://www.w3.org/TR/swbp-n-aryRelations/

Functions. We model functions as views with binding patterns [26], namely:

$$f(\underline{x}, y_1, ..., y_m) \leftarrow B_1, ..., B_n$$

Here, f is the function name, x is the *input variable* (which we underline), $y_1, ..., y_m$ are the *output variables*, and any other variables of the body atoms are *existential variables*. In this paper, we are concerned with *path functions*, where the body atoms are ordered in a sequence $r_1(\underline{x}, x_1), r_2(x_1, x_2), ..., r_n(x_{n-1}, x_n)$, the first variable of the first atom is the input of the plan, the second variable of each atom is the first variable of its successor, and the output variables are ordered in the same way as the atoms.

Example 3.1. *Consider again our example in Figure 1. There are 3 relations names in the database: onAlbum, sang, and relAlbum. The relation relAlbum links a song to a related album. The functions are:*

$$\begin{aligned} \mathit{getAlbum}(\underline{s}, a) &\leftarrow \mathit{onAlbum}(\underline{s}, a) \\ \mathit{getAlbumDetails}(\underline{a}, s, m) &\leftarrow \mathit{onAlbum}^-(\underline{a}, s), \mathit{sang}^-(s, m) \\ \mathit{getRelAlbum}(\underline{s}, a) &\leftarrow \mathit{relAlbum}(\underline{s}, a) \end{aligned}$$

The first function takes as input a song s, and returns as output the album a of the song. The second function takes as input an album a and returns the songs s with their musicians m. The last function returns the related albums of a song.

Execution Plans. Our goal in this work is to study when we can evaluate an atomic query on an instance using a set of path functions, which we will do using *plans*. Formally, a *plan* is a finite sequence $\pi_a(x) = c_1, \ldots, c_n$ of *function calls*, where a is a constant, x is the output variable. Each function call c_i is of the form $f(\underline{\alpha}, \beta_1, \ldots, \beta_n)$, where f is a function name, where the input α is either a constant or a variable occurring in some call in $c_1, \ldots, c_{i-1}$, and where the outputs $\beta_1, \ldots, \beta_n$ are either variables or constants. A *filter* in a plan is the use of a constant in one of the outputs β_i of a function call; if the plan has none, then we call it *unfiltered*. The *semantics* of the plan is the query:

$$q(x) \leftarrow \phi(c_1), \ldots, \phi(c_n)$$

where each $\phi(c_i)$ is the body of the query defining the function f of the call c_i in which we have substituted the constants and variables used in c_i, where we have used fresh existential variables across the different $\phi(c_i)$, and where x is the output variable of the plan.

To *evaluate* a plan on an instance means running the query above. Given an execution plan π_a and a database I, we call $\pi_a(I)$ the answers of the plan on I. In practice, evaluating the plan means calling the functions in the order given by the plan. If a call fails, it can potentially remove one or all answers of the plan. More precisely, for a given instance I, the results $b \in \pi_a(I)$ are precisely the elements b to which we can bind the output variable when matching the semantics of the plan on I. For example, let us consider a function $f(\underline{x}, y) = r(x, y)$ and a plan $\pi_a(x) = f(a, x), f(b, y)$. This plan returns the answer a' on the instance $I = \{r(a, a'), r(b, b')\}$, and returns no answer on $I' = \{r(a, a')\}$.

Example 3.2. *The following is an execution plan for Example 3.1:*

$$\pi_{Jailhouse}(m) = getAlbum(\underline{Jailhouse}, a), getAlbumDetails(\underline{a}, Jailhouse, m)$$

The first element is a function call to getAlbum with the constant Jailhouse as input, and the variable a as output. The variable a then serves as input in the second function call to getAlbumDetails. The plan is shown in Figure 1 on page 1 with an example instance. This plan defines the query:

$$onAlbum(Jailhouse, a), onAlbum^{-}(a, Jailhouse), sang^{-}(Jailhouse, m)$$

For our example instance, we have the embedding:

$$\sigma = \{a = JailhouseRock, m = ElvisPresley\}.$$

Atomic Query Rewriting. Our goal is to determine when a given atomic query $q(x)$ can be evaluated as a plan $\pi_a(x)$. Formally, we say that $\pi_a(x)$ is a *rewriting* (or an *equivalent plan*) of the query $q(x)$ if, for any database instance I satisfying the inclusion dependencies $\mathcal{UID}$, the result of the plan π_a is equal to the result of the query q on I.

4 Problem Statement and Main Results

The goal of this paper is to determine when a query admits a rewriting under the inclusion dependencies. If so, we compute a rewriting. In this section, we present our main high-level results for this task. We then describe in the next section (Section 5) the algorithm that we use to achieve these results, and show in Section 6 our experimental results on an implementation of this algorithm.

Remember that we study *atomic* queries, e.g., $q(x) \leftarrow r(a, x)$, that we study plans on a set $\mathcal{F}$ of path functions, and that we assume that the data satisfy integrity constraints given as a set $\mathcal{UID}$ of *unary inclusion dependencies*. In this section, we first introduce the notion of *non-redundant plans*, which are a specific class of plans that we study throughout the paper; and we then state our results about finding rewritings that are non-redundant plans.

4.1 Non-redundant plans

Our goal in this section is to restrict to a well-behaved subset of plans that are *non-redundant*. Intuitively, a *redundant plan* is a plan that contains function calls that are not useful to get the output of the plan. For example, if we add the function call $getAlbum(m, a')$ to the plan in Example 3.2, then this is a redundant call that does not change the result of $\pi_{Jailhouse}$. We also call *redundant* the calls that are used to remove some of the answers, e.g., for the function $f(\underline{x}, y) = r(x, y)$ and the plan $\pi_a(x) = f(a, x), f(b, y)$ presented before, the second call is redundant because it does not contribute to the output (but can filter out some results). Formally:

Definition 4.1 (Redundant plan). *An execution plan $\pi_a(x)$ is* redundant *if it has no call using the constant a as input, or if it contains a call where none of the outputs is an output of the plan or an input to another call. If the plan does not satisfy these conditions, it is* non-redundant*.*

Non-redundant plans can easily be reformulated to have a more convenient shape: the first call uses the input value as its input, and each subsequent call uses as its input a variable that was an output of the previous call. Formally:

Property 4.2. *The function calls of any non-redundant plan $\pi_a(x)$ can be organized in a sequence $c_0, c_1, \ldots, c_k$ such that the input of c_0 is the constant a, every other call c_i takes as input an output variable of the previous call c_{i-1}, and the output of the plan is in the call c_k.*

Non-redundant plans seem less potent than redundant plans, because they cannot, e.g., filter the outputs of a call based on whether some other call is successful. However, as it turns out, we can restrict our study to non-redundant plans without loss of generality, which we do in the remainder of the paper.

Property 4.3. *For any redundant plan $\pi_a(x)$ that is a rewriting to an atomic query $q(x) \leftarrow r(a, x)$, a subset of its calls forms a non-redundant plan, which is also equivalent to $q(x)$.*

4.2 Result statements

Our main theoretical contribution is the following theorem:

Theorem 4.4. *There is an algorithm which, given an atomic query $q(x) \leftarrow r(a, x)$, a set $\mathcal{F}$ of path function definitions, and a set $\mathcal{UID}$ of UIDs, decides in polynomial time if there exists an equivalent rewriting of q. If so, the algorithm enumerates all the non-redundant plans that are equivalent rewritings of q.*

In other words, we can efficiently decide if equivalent rewritings exist, and when they do, the algorithm can compute them. Note that, in this case, the generation of an equivalent rewriting is *not* guaranteed to be in polynomial time, as the equivalent plans are not guaranteed to be of polynomial size. Also, observe that this result gives a *characterization* of the equivalent non-redundant plans, in the sense that *all* such plans are of the form that our algorithm produces. Of course, as the set of equivalent non-redundant plans is generally infinite, our algorithm cannot actually write down all such plans, but it provides any such plan after a finite time. The underlying characterization of equivalent non-redundant plans is performed via a context-free grammar describing possible paths of a specific form, which we will introduce in the next section.

Our methods can also solve a different problem: given the query, path view definitions, unary inclusion dependencies, and given a candidate non-redundant plan, decide if the plan is correct, i.e., if it is an equivalent rewriting of the query. The previous result does not provide a solution as it produces all non-redundant equivalent plans in some arbitrary order. However, we can show using similar methods that this task can also be decided in polynomial time:

Proposition 4.5. *Given a set of unary inclusion dependencies, a set of path functions, an atomic query $q(x) \leftarrow r(a, x)$ and a non-redundant execution plan π_a, one can determine in PTIME if π_a is an equivalent rewriting of q.*

That proposition concludes the statement of our main theoretical contributions. We describe in the next section the algorithm used to show our main theorem (Theorem 4.4) and used for our experiments in Section 6. The appendix contains the proofs for our theorems.

5 Algorithm

We now present the algorithm used to show Theorem 4.4. The presentation explains at a high level how the algorithm can be implemented, as we did for the experiments in Section 6. However, some formal details of the algorithm are deferred to the appendix, as well as the formal proof.

Our algorithm is based on a characterization of the non-redundant equivalent rewritings as the intersection between a context-free grammar and a regular expression (the result of which is itself a context-free language). The context-free grammar encodes the UID constraints and generates a language of words that intuitively describe forward-backward paths that are guaranteed to exist under the UIDs. As for the regular expression, it encodes the path functions and expresses the legal execution plans. Then, the intersection gets all non-redundant execution plans that satisfy the UIDs. We first detail the construction of the grammar, and then of the regular expression.

5.1 Defining the context-free grammar of forward-backward paths

Our context-free grammar intuitively describes a language of forward-backward paths, which intuitively describe the sequences of relations that an equivalent plan can take to walk away from the input value on an instance, and then walk back to that value, as in our example on Figure 1, to finally use the relation that consists of the query answer: in our example, the plan is *getAlbum(Jailhouse,a), getAlbumDetails(a,Jailhouse,m)*. The grammar then describes all such back-and-forth paths from the input value that are guaranteed to exist thanks to the unary inclusion dependencies that we assumed in $\mathcal{UID}$. Intuitively, it describes such paths in the *chase* by $\mathcal{UID}$ of an answer fact. We now define this grammar, noting that the definition is independent of the functions in $\mathcal{F}$:

Definition 5.1 (Grammar of forward-backward paths). *Given a set of relations $\mathcal{R}$, given an atomic query $q(a, x) \leftarrow r(a, x)$ with $r \in \mathcal{R}$, and given a set of unary inclusion dependencies $\mathcal{UID}$, the* grammar of forward-backward paths *is a context-free grammar $\mathcal{G}_q$, whose language is written $\mathcal{L}_q$, with the non-terminal symbols $S \cup \{L_{r_i}, B_{r_i} \mid r_i \in \mathcal{R}\}$, the terminals $\{r_i \mid r_i \in \mathcal{R}\}$, the start symbol S, and the following productions:*

$$S \to B_r r \tag{5.1}$$
$$S \to B_r r B_{r^-} r^- \tag{5.2}$$
$$\forall r_i, r_j \in \mathcal{R} \text{ s.t. } r_i \rightsquigarrow r_j \text{ in } \mathcal{UID} : B_{r_i} \to B_{r_i} L_{r_j} \tag{5.3}$$
$$\forall r_i \in \mathcal{R} : B_{r_i} \to \epsilon \tag{5.4}$$
$$\forall r_i \in \mathcal{R} : L_{r_i} \to r_i B_{r_i^-} r_i^- \tag{5.5}$$

The words of this grammar describe the sequence of relations of paths starting at the input value and ending by the query relation r, which are guaranteed to exist thanks to the unary inclusion dependencies $\mathcal{UID}$. In this grammar, the B_{r_i}s represent the paths that "loop" to the position where they started, at which we have an outgoing r_i-fact. These loops are either empty (Rule 5.4), are concatenations of loops which may involve facts implied by $\mathcal{UID}$ (Rule 5.3), or may involve the outgoing r_i fact and come back in the reverse direction using r_i^- after a loop at a position with an outgoing r_i^--fact (Rule 5.5).

5.2 Defining the regular expression of possible plans

While the grammar of forward-backward paths describes possible paths that are guaranteed to exist thanks to $\mathcal{UID}$, it does not reflect the set $\mathcal{F}$ of available functions. This is why we intersect it with a regular expression that we will construct from $\mathcal{F}$, to describe the possible sequences of calls that we can perform following the description of non-redundant plans given in Property 4.2.

The intuitive definition of the regular expression is simple: we can take any sequence of relations, which is the semantics of a function in $\mathcal{F}$, and concatenate such sequences to form the sequence of relations corresponding to what the plan retrieves. However, there are several complications. First, for every call, the output variable that we use may not be the last one in the path, so performing the call intuitively corresponds to a prefix of its semantics: we work around this by adding some backward relations to focus on the right prefix when the output variable is not the last one. Second, the last call must end with the relation r used in the query, and the variable that precedes the output variable of the whole plan must not be existential (otherwise, we will not be able to filter on the correct results). Third, some plans consisting of one single call must be handled separately. Last, the definition includes other technicalities that relate to our choice of so-called *minimal filtering plans* in the correctness proofs that we give in the appendix. Here is the formal definition:

Definition 5.2 (Regular expression of possible plans). *Given a set of functions $\mathcal{F}$ and an atomic query $q(x) \leftarrow r(a, x)$, for each function $f : r_1(x_0, x_1), ...r_n(x_{n-1}, x_n)$ of $\mathcal{F}$ and input or output variable x_i, define:*

$$w_{f,i} = \begin{cases} r_1 \dots r_i & \text{if } i = n \\ r_1 \dots r_n r_n^- ... r_{i+1}^- & \text{if } 0 \leq i < n \end{cases}$$

For $f \in \mathcal{F}$ and $0 \leq i < n$, we say that a $w_{f,i}$ is final *when:*

- *the last letter of* $w_{f,i}$ *is* r^-*, or it is* r *and we have* $i > 0$*;*
- *writing the body of* f *as above, the variable* x_{i+1} *is an output variable;*
- *for* $i < n-1$*, if* x_{i+2} *is an output variable, we require that* f *does not contain the atoms:* $r(x_i, x_{i+1}).r^-(x_{i+1}, x_{i+2})$*.*

The regular expression of possible plans is then $P_r = W_0|(W^*W')$*, where:*

- W *is the disjunction over all the* $w_{f,i}$ *above with* $0 < i \leq n$*.*
- W' *is the disjunction over the final* $w_{f,i}$ *above with* $0 < i < n$*.*
- W_0 *is the disjunction over the final* $w_{f,i}$ *above with* $i = 0$*.*

5.3 Defining the algorithm

We can now present our algorithm to decide the existence of equivalent rewritings and enumerate all non-redundant equivalent execution plans when they exist, which is what we use to show Theorem 4.4:
Input: a set of path functions $\mathcal{F}$, a set of relations $\mathcal{R}$, a set of $\mathcal{UID}$ of UIDs, and an atomic query $q(x) \leftarrow r(a, x)$.
Output: a (possibly infinite) list of rewritings.

1. Construct the grammar $\mathcal{G}_q$ of forward-backward paths (Definition 5.1).
2. Construct the regular expression P_r of possible plans (Definition 5.2).
3. Intersect P_r and $\mathcal{G}_q$ to create a grammar $\mathcal{G}$
4. Determine if the language of $\mathcal{G}$ is empty:
   If no, then no equivalent rewritings exist and stop;
   If yes, then continue
5. For each word w in the language of $\mathcal{G}$:
   - For each execution plan $\pi_a(x)$ that can be built from w (intuitively decomposing w using P_r, see appendix for details):
     - For each subset S of output variables of $\pi_a(x)$:
       - If adding a filter to a on the outputs in S gives an equivalent plan, then output the plan (see appendix for how to decide this)

Our algorithm thus decides the existence of an equivalent rewriting by computing the intersection of a context-free language and a regular language and checking if its language is empty. As this problem can be solved in PTIME, the complexity of our entire algorithm is polynomial in the size of its input. The correctness proof of our algorithm (which establishes Theorem 4.4), and the variant required to show Proposition 4.5, are given in the appendix.

6 Experiments

We have given an algorithm that, given an atomic query and a set of path functions, generates all equivalent plans for the query (Section 5). We now compare our approach experimentally to two other methods, Susie [23], and PDQ [3], on both synthetic datasets and real functions from Web services.

6.1 Setup

We found only two systems that can be used to rewrite a query into an equivalent execution plan: Susie [23] and PDQ (Proof-Driven Querying) [3]. We benchmark them against our implementation. All algorithms must answer the same task: given an atomic query and a set of path functions, produce an equivalent rewriting, or claim that there is no such rewriting.

We first describe the Susie approach. Susie takes as input a query and a set of Web service functions and extracts the answers to the query both from the functions and from Web documents. Its rewriting approach is rather simple, and we have reimplemented it in Python. However, the Susie approach is not complete for our task: she may fail to return an equivalent rewriting even when one exists. What is more, as Susie is not looking for equivalent plans and makes different assumptions from ours, the plan that she returns may not be equivalent rewritings (in which case there may be a different plan which is an equivalent rewriting, or no equivalent rewriting at all).

Second, we describe PDQ. The PDQ system is an approach to generating query plans over semantically interconnected data sources with diverse access interfaces. We use the official Java release of the system. PDQ runs the chase algorithm [1] to create a canonical database, and, at the same time, tries to find a plan in that canonical database. If a plan exists, PDQ will eventually find it; and whenever PDQ claims that there is no equivalent plan, then indeed no equivalent plan exists. However, in some cases, the chase algorithm used by PDQ may not terminate. In this case, it is impossible to know whether the query has a rewriting or not. We use PDQ by first running the chase with a timeout, and re-running the chase multiple times in case of timeouts while increasing the search depth in the chase, up to a maximal depth. The exponential nature of PDQ's algorithm means that already very small depths (around 20) can make the method run for hours on a single query.

Our method is implemented in Python and follows the algorithm presented in the previous section. For the manipulation of formal languages, we used pyformlang[4]. Our implementation is available online[5]. All experiments were run on a laptop with Linux, 1 CPU with 4 cores at 2.5GHz, and 16 GB RAM.

6.2 Synthetic Functions

In our first experiments, we consider a set of artificial relations $\mathcal{R} = \{r_1, ..., r_n\}$, and randomly generate path functions up to length 4. Then we tried to find a equivalent plan for each query of the form $r(c, x)$ for $r \in \mathcal{R}$. The set $\mathcal{UID}$ consists of all pairs of relations $r \rightsquigarrow s$ for which there is a function in whose body r^- and s appear in two successive atoms. We made this choice because functions without these UIDs are useless in most cases.

For each experiment that we perform, we generate 200 random instances of the problem, run each system on these instances, and average the results of each

[4] https://pyformlang.readthedocs.io
[5] https://github.com/Aunsiels/query_rewriting

method. Because of the large number of runs, we had to put a time limit of 2 minutes per chase for PDQ and a maximum depth of 16 (so the maximum total time with PDQ for each query is 32 minutes). In practice, PDQ does not strictly abide by the time limit, and its running time can be twice longer. We report, for each experiment, the following numbers:

- Ours: The proportion of instances for which our approach found an equivalent plan. As our approach is proved to be correct, this is the true proportion of instances for which an equivalent plan exists.
- Susie: The proportion of instances for which Susie returned a plan which is actually an equivalent rewriting (we check this with our approach).
- PDQ: The proportion of instances for which PDQ returned an equivalent plan (without timing out): these plans are always equivalent rewritings.
- Susie Requires Assumption: The proportion of instances for which Susie returned a plan, but the returned plan is not an equivalent rewriting (i.e., it is only correct under the additional assumptions made by Susie).
- PDQ Timeout: The proportion of instances for which PDQ timed out (so we cannot conclude whether a plan exists or not).

In all cases, the two competing approaches (Susie and PDQ) cannot be better than our approach, as we always find an equivalent rewriting when one exists, whereas Susie may fail to find one (or return a non-equivalent one), and PDQ may timeout. The two other statistics (Susie Requires Assumption, and PDQ Timeout) denote cases where our competitors fail, which cannot be compared to the performance of our method.

In our first experiment, we limited the number of functions to 15, with 20% of existential variables, and varied the number n of relations. Both Susie and our algorithm run in less than 1 minute in each setting for each query, whereas PDQ may timeout. Figure 2a shows which percentage of the queries can be answered. As expected, when the number of relations increases, the rate of answered queries decreases as it becomes harder to combine functions. Our approach can always answer strictly more queries than Susie and PDQ.

In our next experiment, we fixed the number of relations to 7, the probability of existential variables to 20%, and varied the number of functions. Figure 2b shows the results. As we increase the number of functions, we increase the number of possible function combinations. Therefore, the percentage of answered queries increases both for our approach and for our competitors. However, our approach answers about twice as many queries as Susie and PDQ.

In our last experiment, we fixed the number of relations to 7, the number of functions to 15, and we varied the probability of having an existential variable. Figure 2c shows the results. As we increase the probability of existential variables, the number of possible plans decreases because fewer outputs are available to call other functions. However, the impact is not as marked as before, because we have to impose at least one output variable per function, which, for small functions, results in few existential variables. As Susie and PDQ use these short functions in general, changing the probability did not impact them too much. Still, our approach can answer about twice as many queries as Susie and PDQ.

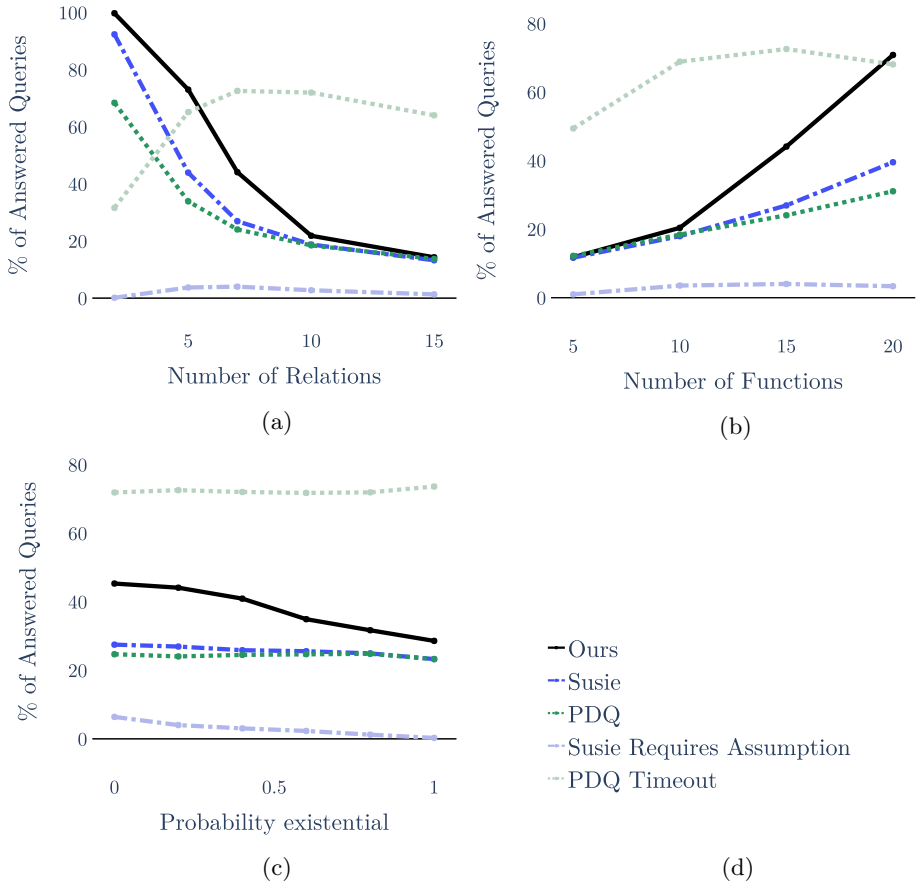

Fig. 2: Percentage of answered queries with varying number of (a) relations, (b) functions, and (c) existential variables; (d) key to the plots.

6.3 Real-World Web Services

We consider the functions of Abe Books (http://search2.abebooks.com), ISBNDB (http://isbndb.com/), LibraryThing (http://www.librarything.com/), and MusicBrainz (http://musicbrainz.org/), all used in [23], and Movie DB (https://www.themoviedb.org) to replace the (now defunct) Internet Video Archive used in [23]. We add to these functions some other functions built by the Susie approach. We group these Web services into three categories: Books, Movies, and Music, on which we run experiments separately. For each category, we manually map all services into the same schema and generate the UIDs as in Section 6.2. Our dataset is available online (see URL above).

The left part of Table 1 shows the number of functions and the number of relations for each Web service. Table 2 gives examples of functions. Some of

Table 1: Web services and results

Web Service	Functions	Relations	Susie	PDQ (timeout)	Ours
Movies	2	8	13%	**25**% (0%)	**25**%
Books	13	28	57%	64% (7%)	**68**%
Music	24	64	22%	22% (25%)	**33**%

Table 2: Examples of real functions

GetCollaboratorsByID(<u>artistId</u>, collab, collabId) ←
  hasId$^-$(artistId,artist), isMemberOf(artist,collab), hasId(collab,collabId)
GetBookAuthorAndPrizeByTitle(<u>title</u>, author, prize) ←
  isTitled$^-$(title, book), wrote$^-$(book,author), hasWonPrize(author,prize)
GetMovieDirectorByTitle(<u>title</u>, director) ←
  isTitled$^-$(title,movie), directed$^-$(movie,director)

them are recursive. For example, the first function in the table allows querying for the collaborators of an artist, which are again artists. This allows for the type of infinite plans that we discussed in the introduction, and that makes query rewriting difficult.

For each Web service, we considered all queries of the form $r(c, x)$ and $r^-(c, x)$, where r is a relation used in a function definition. We ran the Susie algorithm, PDQ, and our algorithm for each of these queries. The runtime is always less than 1 minute for each query for our approach and Susie but can timeout for PDQ. The time limit is set to 30 minutes for each chase, and the maximum depth is set to 16. Table 1 shows the results, similarly to Section 6.2. As in this case, all plans returned by Susie happened to be equivalent plans, we do not include the "Susie Requires Assumption" statistic (it is 0%). Our approach can always answer more queries than Susie and PDQ, and we see that with more complicated problems (like Music), PDQ tends to timeout more often.

In terms of the results that we obtain, some queries can be answered by rather short execution plans. Table 3 shows a few examples. However, our results show that many queries do not have an equivalent plan. In the Music domain, for example, it is not possible to answer $produced(c, x)$ (i.e., to know which albums a producer produced), $hasChild^-(c,x)$ (to know the parents of a person), and $rated^-(c, x)$ (i.e., to know which tracks have a given rating). This illustrates that the services maintain control over the data, and do not allow arbitrary requests.

7 Conclusion

In this paper, we have addressed the problem of finding equivalent execution plans for Web service functions. We have characterized these plans for atomic queries and path functions, and we have given a correct and complete method to find them. Our experiments have demonstrated that our approach can be

Table 3: Example plans

Query	Execution Plan
released	GetArtistInfoByName, GetReleasesByArtistID, GetArtistInfoByName, GetTracksByArtistID, GetTrackInfoByName, GetReleaseInfoByName
published	GetPublisherAuthors, GetBooksByAuthorName
actedIn	GetMoviesByActorName, GetMovieInfoByName

applied to real-world Web services and that its completeness entails that we always find plans for more queries than our competitors. All experimental data, as well as all code, is available at the URL given in Section 6. We hope that our work can help Web service providers to design their functions, and users to query the services more efficiently. For future work, we aim to broaden our results to non-path functions. We also intend to investigate connections between our theoretical results and the methods by Benedikt et al. [2], in particular possible links between our techniques and those used to answer regular path queries under logical constraints [5].

Acknowledgements. Partially supported by the grants ANR-16-CE23-0007-01 ("DICOS") and ANR-18-CE23-0003-02 ("CQFD").

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. Michael Benedikt, Julien Leblay, Balder ten Cate, and Efthymia Tsamoura. *Generating Plans from Proofs: The Interpolation-based Approach to Query Reformulation*. Synthesis Lectures on Data Management. Morgan & Claypool, 2016.
3. Michael Benedikt, Julien Leblay, and Efthymia Tsamoura. PDQ: Proof-driven query answering over web-based data. *VLDB*, 7(13), 2014.
4. Michael Benedikt, Julien Leblay, and Efthymia Tsamoura. Querying with access patterns and integrity constraints. *PVLDB*, 8(6), 2015.
5. Meghyn Bienvenu, Magdalena Ortiz, and Mantas Simkus. Regular path queries in lightweight description logics: Complexity and algorithms. *JAIR*, 53, 2015.
6. A. Bozzon, M. Brambilla, and S. Ceri. Answering search queries with crowdsearcher. In *WWW*, 2012.
7. Andrea Calì, Diego Calvanese, and Davide Martinenghi. Dynamic query optimization under access limitations and dependencies. In *J. UCS*, 2009.
8. Andrea Calì and Davide Martinenghi. Querying data under access limitations. In *ICDE*, 2008.
9. S. Ceri, A. Bozzon, and M. Brambilla. The anatomy of a multi-domain search infrastructure. In *ICWE*, 2011.
10. Namyoun Choi, Il-Yeol Song, and Hyoil Han. A survey on ontology mapping. In *SIGMOD Rec.*, 2006.
11. Daniel Deutch and Tova Milo. *Business Processes: A Database Perspective*. Synthesis Lectures on Data Management. Morgan & Claypool, 2012.
12. Alin Deutsch, Bertram Ludäscher, and Alan Nash. Rewriting queries using views with access patterns under integrity constraints. In *Theor. Comput. Sci.*, 2007.

13. Oliver M. Duschka and Michael R. Genesereth. Answering recursive queries using views. In *PODS*, 1997.
14. Oliver M. Duschka, Michael R. Genesereth, and Alon Y. Levy. Recursive query plans for data integration. In *J. Log. Program.*, 2000.
15. Daniela Florescu, Alon Y. Levy, Ioana Manolescu, and Dan Suciu. Query optimization in the presence of limited access patterns. In *SIGMOD*, 1999.
16. Tomasz Gogacz and Jerzy Marcinkowski. Red spider meets a rainworm: Conjunctive query finite determinacy is undecidable. In *SIGMOD*, 2016.
17. Alon Y. Halevy. Answering queries using views: A survey. In *VLDB J.*, 2001.
18. John E Hopcroft and 1942 Ullman, Jeffrey D. *Introduction to automata theory, languages, and computation.* Reading, Mass. : Addison-Wesley, 1979.
19. Maria Koutraki, Dan Vodislav, and Nicoleta Preda. Deriving intensional descriptions for web services. In *CIKM*, 2015.
20. David L. Martin, Massimo Paolucci, Sheila A. McIlraith, Mark H. Burstein, Drew V. McDermott, Deborah L. McGuinness, Bijan Parsia, Terry R. Payne, Marta Sabou, Monika Solanki, Naveen Srinivasan, and Katia P. Sycara. Bringing semantics to web services: The OWL-S approach. In *SWSWPC*, 2004.
21. Alan Nash and Bertram Ludäscher. Processing unions of conjunctive queries with negation under limited access patterns. In *EDBT*, 2004.
22. N. Preda, G. Kasneci, F. M. Suchanek, T. Neumann, W. Yuan, and G. Weikum. Active Knowledge : Dynamically Enriching RDF Knowledge Bases by Web Services. In *SIGMOD*, 2010.
23. Nicoleta Preda, Fabian M. Suchanek, Wenjun Yuan, and Gerhard Weikum. SUSIE: Search Using Services and Information Extraction. In *ICDE*, 2013.
24. Ken Q. Pu, Vagelis Hristidis, and Nick Koudas. Syntactic rule based approach to Web service composition. In *ICDE*, 2006.
25. Bastian Quilitz and Ulf Leser. Querying distributed RDF data sources with SPARQL. In *ESWC*, 2008.
26. Anand Rajaraman, Yehoshua Sagiv, and Jeffrey D. Ullman. Answering queries using templates with binding patterns. In *PODS*, 1995.
27. Jinghai Rao, Peep Küngas, and Mihhail Matskin. Logic-based web services composition: From service description to process model. In *ICWS*, 2004.
28. Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. Fedx: Optimization techniques for federated query processing on linked data. In *ISWC*, 2011.
29. Utkarsh Srivastava, Kamesh Munagala, Jennifer Widom, and Rajeev Motwani. Query optimization over web services. In *VLDB*, 2006.
30. OASIS Standard. Web services business process execution language. https://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf, April 2007.
31. Mohsen Taheriyan, Craig A. Knoblock, Pedro A. Szekely, and José Luis Ambite. Rapidly integrating services into the linked data cloud. In *ISWC*, 2012.
32. Snehal Thakkar, José Luis Ambite, and Craig A. Knoblock. Composing, optimizing, and executing plans for bioinformatics web services. In *VLDB J.*, 2005.
33. WSML working group. WSML language reference. http://www.wsmo.org/wsml/, 2008.

A Proofs on Non-Redundant Plans (Section 4.1)

In this first section of the appendix, we give proofs pertaining to non-redundant plans (Section 4.1). We introduce in particular the notion of *well-filtering plans* (Section A.2), which will be useful later.

The next section of the appendix (Appendix B) gives a high-level presentation of key technical results about *minimal filtering plans* and *capturing languages*. These claims are then proved in Appendix C. Last, we give in Appendix D the proofs of the missing details of our main claims (Section 4) and of our algorithm (Section 5).

A.1 Proof of the Structure of Non-Redundant Plans (Property 4.2)

Property 4.2. *The function calls of any non-redundant plan $\pi_a(x)$ can be organized in a sequence $c_0, c_1, \ldots, c_k$ such that the input of c_0 is the constant a, every other call c_i takes as input an output variable of the previous call c_{i-1}, and the output of the plan is in the call c_k.*

By definition of a non-redundant plan, there is an atom using the constant a as input. Let us call this atom c_0. Let us then define the sequence $c_0, c_1, \ldots, c_i$, and let us assume that at some stage we are stuck, i.e., we have chosen a call c_i such that none of the output variables of c_i are used as input to another call. If the output of the plan is not in c_i, then c_i witnesses that the plan is redundant. Otherwise, the output of the plan is in c_i. If we did not have $i = k$, then any of the calls not in $c_0, c_1, \ldots, c_i$ witness that the plan is redundant. So we have $i = k$, and we have defined the sequence $c_0, c_1, \ldots, c_k$ as required.

A.2 Well-Filtering Plans

In this subsection, we introduce *well-filtering plans*, which are used both to show that we can always restrict to non-redundant plans (Property 4.3, showed in the next appendix section) and for the correctness proof of our algorithm. We then show a result (Lemma A.2) showing that we can always restrict our study to well-filtering plans.

Let us first recall the notion of the *chase* [1]. The chase of an instance I by a set $\mathcal{UID}$ of unary inclusion dependencies (UIDs) is a (generally infinite) instance obtained by iteratively solving the violations of $\mathcal{UID}$ on I by adding new facts. In particular, if I already satisfies $\mathcal{UID}$, then the chase of I by $\mathcal{UID}$ is equal to I itself. The chase is known to be a *canonical database* in the sense that it satisfies precisely the queries that are true on all completions of I to make it satisfy $\mathcal{UID}$. We omit the formal definition of the chase and refer the reader to [1] for details about this construction. We note the following property, which can be achieved whenever $\mathcal{UID}$ is closed under UID implication, and when we do the so-called *restricted* chase which only solves the UID violations that are not already solved:

Property 1. Let f be a single fact, and let I be the instance obtained by applying the chase on f. Then for each element c of I_0, for each relation $r \in \mathcal{R}$, there is at most one fact of I_0 where c appears in the first position of a fact for relation r.

Remember now that that plans can use *filters*, which allow us to only consider the results of a function call where some variable is assigned to a specific constant. In this section, we show that, for any plan π_a, the only filters required are on the constant a. Further, we show that they can be applied to a single well-chosen atom.

Definition A.1 (Well-Filtering Plan). *Let $q(x) \leftarrow r(a, x)$ be an atomic query. An execution plan $\pi_a(x)$ is said to be* well-filtering for $q(x)$ *if all filters of the plan are on the constant a used as input to the first call and the semantics of π_a contains at least an atom $r(a, x)$ or $r^-(x, a)$, where x is the output variable.*

We can then show :

Lemma A.2. *Given an atomic query $q(a, x) \leftarrow r(a, x)$ and a set of inclusion dependencies $\mathcal{UID}$, any equivalent rewriting of q must be well-filtering.*

Proof. We first prove the second part. We proceed by contradiction. Assume that there is a non-redundant plan $\pi_a(x)$ which is an equivalent rewriting of $q(a, x)$ and which contains a constant $b \neq a$. By Property 4.2, the constant b is not used as the input to a call (this is only the case of a, in atom c_0), so b must be used as an output filter in π_a. Now, consider the database $I = \{r(a, a')\}$, and let I^* be the result of applying the chase by $\mathcal{UID}$ to I. The result of the query q on I^* is a', and I^* satisfies $\mathcal{UID}$ by definition, however b does not appear in I^* so π_a does not return anything on I^* (its semantics cannot have a match), a contradiction.

We now prove the first part of the lemma. We use the form of Property 4.2. If we separate the body atoms where a is an argument from those where both arguments are variables, we can write: $q'(a, x) \leftarrow A(a, x_1, x_2, \dots x_n), B(x_1, x_2, \dots x_n)$ where $A(a, x_1, x_2, \dots x_n) \leftarrow r_1(a, x_1), \dots r_n(a, x_n)$ (if we have an atom $r_i(x, a)$ we transform it into $r_i^-(a, x)$) and a does not appear as argument in any of the body atoms of $B(x_1, x_2, \dots x_n)$. By contradiction, assume that we have $r_i \neq r$ for all $1 \leq i \leq n$.

Let I_0 be the database containing the single fact $r(a, b)$ and consider the database I_0^* obtained by chasing the fact $r(a, b)$ by the inclusion dependencies in $\mathcal{UID}$, creating a new value to instantiate every missing fact. Let $I_1^* = I_0^* \cup \{r(a_1, b_1)\} \cup \{r_i(a_1, c_i) \mid r_i(a, c_i) \in I_0^* \wedge r_i \neq r\} \cup \{r_i(b_1, c_i) \mid r_i(b, c_i) \in I_0^* \wedge r_i \neq r^-\}$. By construction, I_1^* satisfies $\mathcal{UID}$. Also, we have that $\forall r_i \neq r$, $r_i(a, c_i) \in I_1^* \Leftrightarrow r_i(a_1, c_i) \in I_1^*$. Hence, we have that $A(a, x_1, x_2, \dots x_n)(I_1^*) = A(a_1, x_1, x_2, \dots x_n)(I_1^*)$. Then, given that $B(x_1, x_2, \dots x_n)$ does not contain a nor a_1, we have that $q'(a_1, x)(I_1^*) = q'(a, x)(I_1^*)$. From the hypothesis we also have that $q'(a, x)(I_1^*) = q(a, x)(I_1^*)$ and $q'(a_1, x)(I_1^*) = q(a_1, x)(I_1^*)$. This implies that $q(a_1, x)(I_1^*) = q(a, x)(I_1^*)$. Contradiction. □

A.3 Proof that we Can Restrict to Non-Redundant Plans (Property 4.3)

We can now prove the property claiming that it suffices to study non-redundant plans. Recall its statement:

Property 4.3. *For any redundant plan $\pi_a(x)$ that is a rewriting to an atomic query $q(x) \leftarrow r(a, x)$, a subset of its calls forms a non-redundant plan, which is also equivalent to $q(x)$.*

In what follows, we write $q(a, x)$ instead of $q(x)$ to clarify the inner constant. Let $\pi_a(x)$ be an equivalent plan. From Lemma A.2, we have that its semantics contains a body atom $r(a, x)$ or $r^-(x, a)$. Hence, there is a call c such that $r(a, x)$ or $r^-(x, a)$ appear in its semantics. From the definition of plans, and similarly to the proof of Property 4.2, there is a chain of calls $c_1, c_2, \dots c_k$ such that c_1 takes a constant as input, $c_k = c$, and for every two consecutive calls c_i and c_{i+1}, with $i \in \{1, \dots k-1\}$, there is a variable α such that α is an output variable for c_i and an output variable for c_{i+1}. From Lemma A.2, we have that for all the calls that take a constant as input, the constant is a. Hence, the input of c_1 is a. Let $\pi'_a(x)$ be the plan consisting of the calls $c_1, c_2, \dots c_k = c$. Note that c ensures that $r(a, x)$ or $r^-(x, a)$ appear in the semantics of $\pi'_a(x)$.

We first notice that by construction $\pi'_a(x)$ is non-redundant. Now, if we consider the semantics of a plan as a set of body atoms, the semantics of $\pi'_a(x)$ is contained in the semantics of $\pi_a(x)$. Hence, we have $\forall I, \pi_a(x)(I) \subseteq \pi'_a(x)(I)$. As $\pi_a(x)$ is equivalent to $q(x) \leftarrow r(a, x)$, $\forall I$, we have $\pi_a(x)(I) = q(x)(I)$. As $\pi'_a(x)$ contains $r(a, x)$, $\pi'_a(x)(I) \subseteq q(x)(I)$. So, $\forall I, q(x)(I) = \pi_a(x)(I) \subseteq \pi'_a(x)(I) \subseteq q(x)(I)$. Hence, all the inclusions are equalities, and indeed $\pi'_a(x)$ is also equivalent to the query under $\mathcal{UID}$. This concludes the proof.

B Capturing Languages

In this section, we give more formal details on our approach, towards a proof of Theorem 4.4 and Proposition 4.5. We will show that we can restrict ourselves to a class of execution plans called *minimal filtering plans* which limit the possible filters in an execution plan. Finally, we will define the notion of *capturing language* and show that the language $\mathcal{L}_q$ defined in Section 5 is capturing (Theorem B.11); and define the notion of a language *faithfully representing plans* and show that the language of the regular expression P_r faithfully represents plans (Theorem B.13). This appendix gives a high-level overview and states the theorem; the next appendix (Appendix C) contains proofs for the present appendix; and the last appendix (Appendix D) contains the proofs of the claims made in Sections 4 and 5.

B.1 Minimal Filtering Plans

Remember the definition of well-filtering plans (Definition A.1). We now simplify even more the filters that should be applied to an equivalent plan, to limit ourselves to a single filter, by introducing *minimal filtering plans*.

Definition B.1 (Minimal Filtering Plan). *Given a well-filtering plan $\pi_a(x)$ for an atomic query $q(a, x) \leftarrow r(a, x)$, let the minimal filtering plan associated to $\pi_a(x)$ be the plan $\pi'_a(x)$ that results from removing all filters from $\pi_a(x)$ and doing the following:*

- *We take the greatest possible call c_i of the plan, and the greatest possible output variable x_j of call c_i, such that adding a filter on a to variable x_j of call c_i yields a well-filtering plan, and define $\pi'_a(x)$ in this way.*
- *If this fails, i.e., there is no possible choice of c_i and x_j, then we leave $\pi_a(x)$ as-is, i.e., $\pi'_a(x) = \pi_a(x)$.*

Note that, in this definition, we assume that the atoms in the semantics of each function follow the order in the definition of the path function. Also, note that the minimal filtering plan $\pi'_a(x)$ associated to a well-filtering plan is always itself well-filtering. This fact is evident if the first case in the definition applies, and in the second case, given that $\pi_a(x)$ was itself well-filtering, the only possible situation is when the first atom of the first call of $\pi_a(x)$ was an atom of the form $r(a, x)$, with a being the input element: otherwise $\pi_a(x)$ would not have been well-filtering. So, in this case, $\pi'_a(x)$ is well-filtering. Besides, note that, when the well-filtering plan π_a is non-redundant, then this is also the case of the minimal filtering plan π_a^{min} because the one filter that we may add is necessarily at an output position of the last call.

Finally, note that a well-filtering plan is not always equivalent to the minimal filtering plan, as removing the additional filters can add some results. However, one can easily check if it is the case or not. This theorem is proven in Appendix C.1.

Theorem B.2. *Given a query $q(x) \leftarrow r(a, x)$, a well-filtering plan π_a, the associated minimal filtering plan π_a^{min} and unary inclusion dependencies:*

- *If π_a^{min} is not equivalent to q, then neither is π_a.*
- *If π_a^{min} is equivalent to q, then we can determine in polynomial time if π_a is equivalent to π_a^{min}*

This theorem implies that, when the query has a rewriting as a well-filtering plan, then the corresponding minimal filtering plan is also a rewriting:

Corollary B.3. *Given unary inclusion dependencies, if a well-filtering plan is a rewriting for an atomic query q, then it is equivalent to the associated minimal filtering plan.*

Proof. This is the contrapositive of the first point of the theorem: if π_a is equivalent to q, then so in π_a^{min}, hence π_a and π_a^{min} are then equivalent. □

For that reason, to study equivalent rewritings, we will focus our attention on minimal filtering plans: Theorem B.3 can identify other well-filtering plans that are rewritings, and we know by Lemma A.2 that plans that are not well-filtering cannot be rewritings.

B.2 Path Transformations

We now show how to encode minimal filtering plans as words over an alphabet whose letters are the relation names in $\mathcal{R}$. The key is to rewrite the plan so that its semantics is a path query of the following form:

Here is the formal notion of a *path query*:

Definition B.4. *A **path query** is a query of the form*

$$q_a(x_i) \leftarrow r_1(a, x_1), r_2(x_1, x_2), ..., r_n(x_{n-1}, x_n)$$

*where a is a constant, x_i is the output variable, each x_j except x_i is either a variable or the constant a, and $1 \leq i \leq n$. The sequence of relations $r_1...r_n$ is called the **skeleton** of the query.*

We formalize as follows the transformation that transforms plans into path queries. We restrict it to non-redundant minimal filtering plans to limit the number of filters that we have to deal with:

Definition B.5 (Path Transformation). *Let $\pi_a(x)$ be a non-redundant minimal filtering execution plan and $\mathcal{R}$ a set of relations. We define the path transformation of $\pi_a(x)$, written $\mathcal{P}'(\pi_a)$, the transformation that maps the plan π_a to a path query $\mathcal{P}'(\pi_a)$ obtained by applying the following steps:*

1. *Consider the sequence of calls $c_0, c_1, ..., c_k$ as defined in Property 4.2, removing the one filter to element a if it exists.*
2. *For each function call $c_i(\underline{y_1}, y_{i_1}..., y_{i_j}, ...y_{i_n}) = r_1(y_1, y_2), ..., r_k(y_k, y_{k+1}), ..., r_m(y_m, y_{m+1})$ in π_a with $1 < i_1 < ... < i_n < m+1$, such that y_{i_j} is the output used as input by the next call or is the output of the plan, we call the* sub-semantics associated to c_i *the query: $r_1...r_m.r_m^-...r_{i_j}^-(y_1, ..., y_{i_j-1}, y'_{i_j}, ..., y'_m, y_{m+1}, ..., y_{i_j})$, where $y'_{i_j}, ..., y'_m$ are new variables. We do nothing if $i_j = m+1$.*
3. *Concatenate the sub-semantics associated to the calls in the order of the sequence of calls. We call this new query the* path semantics.
4. *There are two cases:*
   - *If the semantics of π_a contains the atom $r(a, x)$ (either thanks to a filter to the constant a on an output variable or thanks to the first atom of the first call with a being the input variable), then this atom must have been part of the semantics of the last call (in both cases). The sub-semantics of the last call is therefore of the form $\ldots, r(x_a, x'), r_2(x', x_2), \ldots, r_n(x_{n-1}, x_n), r_n^-(x_n, x_{n-1}), \ldots, r_2^-(x_2, x)$, in which x_a was the variable initially filtered to a (or was the input to the plan, in which case it is still the constant a) and we append the atom $r^-(x, a)$ with a filter on a, where x is the output of the path semantics.*
   - *Otherwise, the semantics of π_a contains an atom $r^-(x, a)$, then again it must be part of the last call whose sub-semantics looks like $\ldots, r^-(x', x'_2), r_2(x'_2, x'_3), ..., r_n(x'_{n-1}, x_n), r_n^-(x_n, x_{n-1}), ..., r(x_1, x)$, in which x'_2 was the variable initially filtered to a, and we replace the last variable x_1 by a, with x being the output of the path semantics.*

We add additional atoms in the last point to ensure that the filter on the last output variable is always on the last atom of the query. Notice that the second point relates to the words introduced in Definition 5.2.

The point of the above definition is that, once we have rewritten a plan to a path query, we can easily see the path query as a word in $\mathcal{R}^*$ by looking at the skeleton. Formally:

Definition B.6 (Full Path Transformation). *Given a non-redundant minimal filtering execution plan, writing $\mathcal{R}$ for the set of relations of the signature, we denote by $\mathcal{P}(\pi_a)$ the word over $\mathcal{R}$ obtained by keeping the skeleton the path query $\mathcal{P}'(\pi_a)$ and we call it the* full path transformation.

Note that this loses information about the filters, but this is not essential.

Example B.7. *Let us consider the two following path functions:*

$$f_1(x, y) = s(x, y), t(y, z)$$
$$f_2(x, y, z) = s^-(x, y), r(y, z), u(z, z')$$

The considered atomic query is $q(x) \leftarrow r(a, x)$. We are given the following non-redundant minimal filtering execution plan:

$$\pi_a(x) = f_1(a, y), f_2(y, a, x)$$

We are going to apply the path transformation to π_a. Following the different steps, we have:

1. *The functions calls without filters are:*

$$c_0(a, y) = s(a, y), t(y, z)$$
$$c_1(y, z, x) = s^-(y, z), r(z, x), u(x, z_1)$$

2. *The sub-semantics associated to each function call are:*
   - *For $c_0 : s(a, y'), t(y', z), t^-(z, y)$*
   - *For $c_1 : s^-(y, z), r(z, x'), u(x', z_1), u^-(z_1, x)$*
3. *The path semantics obtained after the concatenation is:*

$$s(a, y'), t(y', z), t^-(z, y), s^-(y, z), r(z, x'), u(x', z_1), u^-(z_1, x)$$

4. *The semantics of π_a contained $r(a, x)$, so add the atom $r^-(x, a)$ to the path semantics.*

At the end of the path transformation, we get

$$\mathcal{P}'(\pi_a) = s(a, y'), t(y', z), t^-(z, y), s^-(y, z), r(z, x'), u(x', z_1), u^-(z_1, x), r^-(x, a)$$

and:

$$\mathcal{P}(\pi_a) = s, t, t^-, s^-, r, u, u^-, r^-$$

This transformation is not a bijection, meaning that possibly multiple plans can generate the same word:

Example B.8. *Consider three path functions:*

- $f_1(x, y) = s(x, y), t(y, z)$,
- $f_2(x, y, z) = s^-(x, y)r(y, z)$,
- $f_3(x, y, z) = s(x, x_0), t(x_0, x_1), t^-(x_1, x_2), s^-(x_2, x_3), r(x_3, y), r^-(y, z)$,

The execution plan $\pi_a^1(x) = f_1(a, y), f_2(y, a, x)$ then has the same image by the path transformation than the execution plan $\pi_a^2(x) = f_3(a, a, x)$.

However, it is possible to efficiently reverse the path transformation whenever an inverse exists. We show this in Appendix C.2.

Property B.9. *Given a word w in $\mathcal{R}^*$, a query $q(x) \leftarrow r(a, x)$ and a set of path functions, it is possible to know in polynomial time if there exists a non-redundant minimal filtering execution plan π_a such that $\mathcal{P}(\pi_a) = w$. Moreover, if such a π_a exists, we can compute one in polynomial time, and we can also enumerate all of them (there are only finitely many of them).*

B.3 Capturing Language

The path transformation gives us a representation of a plan in $\mathcal{R}^*$. In this section, we introduce our main result to characterize *minimal filtering plans*, which are atomic equivalent rewritings based on languages defined on $\mathcal{R}^*$. First, thanks to the path transformation, we introduce the notion of *capturing language*, which allows us to capture equivalent rewritings using a language defined on $\mathcal{R}^*$.

Definition B.10 (Capturing Language). *Let $q(x) \leftarrow r(a, x)$ be an atomic query. The language Λ_q over $\mathcal{R}^*$ is said to be a* capturing language *for the query q (or we say that Λ_q* captures *q) if for all non-redundant minimal filtering execution plans $\pi_a(x)$, we have the following equivalence: π_a is an equivalent rewriting of q iff we have $\mathcal{P}(\pi_a) \in \Lambda_q$.*

Note that the definition of capturing language does not forbid the existence of words $w \in \Lambda_q$ that are not in the preimage of $\mathcal{P}$, i.e., words for which there does not exist a plan π_a such that $P(\pi_a) = w$. We will later explain how to find a language that is a subset of the image of the transformation $\mathcal{P}$, i.e., a language which *faithfully represents plans*.

Our main technical result, which is used to prove Theorem 4.4, is that we have a context-free grammar whose language captures q: specifically, the grammar $\mathcal{G}_q$ (Definition 5.1):

Theorem B.11. *Given a set of unary inclusion dependencies, a set of path functions, and an atomic query q, the language $\mathcal{L}_q$ captures q.*

B.4 Faithfully representing plans

We now move on to the second ingredient that we need for our proofs: we need a language which *faithfully represents plans*:

Definition B.12. *We say that a language $\mathcal{K}$* faithfully represents plans *(relative to a set $\mathcal{F}$ of path functions and an atomic query $q(x) \leftarrow r(a, x)$) if it is a language over $\mathcal{R}$ with the following property: for every word w over $\mathcal{R}$, we have that w is in $\mathcal{K}$ iff there exists a minimal filtering non-redundant plan π_a such that $\mathcal{P}(\pi_a) = w$.*

We now show the following about the language of our regular expression P_r of possible plans as defined in Definition 5.2.

Theorem B.13. *Let $\mathcal{F}$ be a set of path functions, let $q(x) \leftarrow r(a, x)$ be an atomic query, and define the regular expression P_r as in Definition 5.2. Then the language of P_r faithfully represents plans.*

Theorems B.11 and B.13 will allow us to deduce Theorem 4.4 and Proposition 4.5 from Section 4, as explained in Appendix D.

C Proofs for Appendix B

Let us first define some notions used throughout this appendix. Recall the definition of a *path query* (Definition B.4) and of its skeleton. We sometimes abbreviate the body of the query as $r_1...r_n(\alpha, x_1...x_n)$. We use the expression *path query with a filter* to refer to a path query where a body variable other than α is replaced by a constant. For example, in Figure 1, we can have the path query:

$$q(m, a) \leftarrow \text{sang}(m, s), \text{onAlbum}(s, a)$$

which asks for the singers with their albums. Its skeleton is *sang.onAlbum*.

Towards characterizing the path queries that can serve as a rewriting, it will be essential to study *loop queries*:

Definition C.1 (Loop Query). *We call **loop query** a query of the form: $r_1...r_n(a, a) \leftarrow r_1(a, x_1)...r_n(x_{n-1}, a)$ where a is a constant and $x_1, x_2, ..., x_{n-1}$ are variables such that $x_i = x_j \Leftrightarrow i = j$.*

With these definitions, we can show the results claimed in Appendix B.

C.1 Proof of Theorem B.2

Theorem B.2. *Given a query $q(x) \leftarrow r(a, x)$, a well-filtering plan π_a, the associated minimal filtering plan π_a^{min} and unary inclusion dependencies:*

- *If π_a^{min} is not equivalent to q, then neither is π_a.*
- *If π_a^{min} is equivalent to q, then we can determine in polynomial time if π_a is equivalent to π_a^{min}*

First, let us show the first point. By Definition A.1, we have that π_a contains $r(a, x)$ or $r^-(x, a)$. Let us suppose that π_a is equivalent to q. Let I be the database obtained by taking the one fact $r(a, b)$ and chasing by $\mathcal{UID}$. We know that the semantics of π_a has a binding returning b as an answer. We first argue that π_a^{min} also returns this answer. As π_a^{min} is formed by removing all filters from π_a and then adding possibly a single filter, we only have to show this for the case where we have indeed added a filter. But then the added filter ensures that π_a^{min} is well-filtering, so it creates an atom $r(a, x)$ or $r^-(x, a)$ in the semantics of π_a^{min}, so the binding of the semantics of π_a that maps the output variable to b is also a binding of π_a^{min}.

We then argue that π_a^{min} does not return any other answer. In the first case, as π_a^{min} is well-filtering, it cannot return any different answer than b on I. In the second case, we know by the explanation after Definition A.1 that π_a^{min} is also well-filtering, so the same argument applies. Hence, π_a^{min} is also equivalent to q, which establishes the first point.

Let us now show the more challenging second point. We assume that π_a^{min} is equivalent to q. Recall the definition of a loop query (Definition C.1) and the grammar $\mathcal{G}_q$ defined in Definition 5.1, whose language we denoted as $\mathcal{L}_q$. We first show the following property:

Property C.2. *A loop query $r_1...r_n(a, a)$ is true on all database instances satisfying the unary inclusion dependencies $\mathcal{UID}$ and containing a tuple $r(a, b)$, iff there is a derivation tree in the grammar such that $B_r \xrightarrow{*} r_1...r_n$.*

Proof. We first show the backward direction. The proof is by structural induction on the length of the derivation. We first show the base case. If the length is 0, its derivation necessarily uses Rule 5.4, and the query $\epsilon(a, a)$ is indeed true on all databases.

We now show the induction step. Suppose we have the result for all derivations up to a length of $n-1$. We consider a derivation of length $n > 0$. Let I be a database instance satisfying the inclusion dependencies $\mathcal{UID}$ and containing the fact $r(a, b)$. Let us consider the rule at the root of the derivation tree. It can only be Rule 5.3. Indeed, Rule 5.4 only generates words of length 0. So, the first rule applied was Rule 5.3 $B_r \to B_r L_{r_i}$ for a given UID $r \rightsquigarrow r_i$. Then we have two cases.

The first case is when the next B_r does not derive ϵ in the derivation that we study. Then, there exists $i \in \{2, \ldots, n-1\}$ such that $B_r \xrightarrow{*} r_1 \ldots r_{i-1}$ and $L_{r_i} \xrightarrow{*} r_i \ldots r_n$ (L_{r_i} starts by r_i). From the induction hypothesis we have that $r_1 \ldots r_{i-1}(a, a)$ has an embedding in I, and so has $r_i \ldots r_n(a, a)$. Indeed, we have $B_{r_i} \to L_{r_i} \xrightarrow{*} r_i \ldots r_n$ (as trivially $r_j \rightsquigarrow r_j$) and I contains the tuple $r_i(a, c)$ for some constant c (because we have $r \rightsquigarrow r_i$). Hence, $r_1...r_n(a, a)$ is true. This shows the first case of the induction step.

We now consider the case where the next B_{r_i} derives ϵ. Note that, as $r(a, b) \in I$, there exists c such that $r_i(a, c) \in I$. The next rule in the derivation is $L_{r_i} \to r_i B_{r_i^-} r_i^-$, then $B_{r_i^-} \xrightarrow{*} r_2 \ldots r_{n-1}$, and $r_1 = r_i$ and $r_n = r_i^-$. By applying the induction hypothesis, we have that $r_2 \ldots r_{n-1}(c, c)$ has an embedding in I. Now,

given that $r_1(a,c) \in I$ and $r_n(c,a) \in I$ we can conclude that $r_1 \dots r_n(a,a)$ has an embedding in I. This establishes the first case of the induction step. Hence, by induction, we have shown the backward direction of the proof.

We now show the forward direction. Let I_0 be the database containing the single fact $r(a,b)$ and consider the database I_0^* obtained by chasing the fact $r(a,b)$ by the inclusion dependencies in $\mathcal{UID}$, creating a new null to instantiate every missing fact. This database is generally infinite, and we consider a tree structure on its domain, where the root is the element a, the parent of b is a, and the parent of every null x is the element that occurs together with x in the fact where x was introduced. Now, it is a well-known fact of database theory [1] that a query is true on every superinstance of I_0 satisfying $\mathcal{UID}$ iff that query is true on the chase I_0^* of I_0 by $\mathcal{UID}$. Hence, let us show that all loop queries $r_1 \dots r_n(a,a)$, which hold in I_0^* are the ones that can be derived from B_r.

We show the claim again by induction on the length of the loop query. More precisely, we want to show that, for all $n \geq 0$, for a loop query $r_1 ... r_n(a,a)$ which is true on all database instances satisfying the unary inclusion dependencies $\mathcal{UID}$ and containing a tuple $r(a,b)$, we have:

1. $B_r \xrightarrow{*} r_1 ... r_n$
2. For a match of the loop query on I_0^*, if no other variable than the first and the last are equal to a, then we have: $L_{r_1} \xrightarrow{*} r_1 ... r_n$

If the length of the loop query is 0, then it could have been derived by the Rule 5.4. The length of the loop query cannot be 1 as for all relations r', the query $r'(a,a)$ is not true on all databases satisfying the UIDs and containing a tuple $r(a,b)$ (for example it is not true on I_0^*).

Let us suppose the length of the loop query is 2 and let us write the loop query as $r_1(a,x), r_2(x,a)$ and let $r_1(a,c), r_2(c,a)$ be a match on I_0^*. The fact $r_1(a,c)$ can exist on I_0^* iff $r \rightsquigarrow r_1$. In addition, due to the tree structure of I_0^*, we must have $r_2 = r_1^-$. So, we have $B_r \to L_{r_1} \to r_1 B_{r_1^-} r_1^- \to r_1^-$ and we have show the two points of the inductive claim.

We now suppose that the result is correct up to a length $n-1$ ($n > 2$), and we want to prove that it is also true for a loop query of length n.

Consider a match $r_1(a,a_1), r_2(a_1,a_2), \dots, r_{n-1}(a_{n-2},a_{n-1}), r_n(a_{n-1},a_n)$ of the loop query. Either there is some i such that $a_i = a$, or there is none. If there is at least one, then let us cut the query at all positions where the value of the constant is a. We write the binding of the loop queries on I_0^* : $(r_{i_0} \dots r_{i_1})(a,a).(r_{i_1+1} \dots r_{i_2})(a,a) ... r_{i_{k-1}+1} \dots r_{i_k}(a,a)$ (where $1 = i_0 < i_1 < ... < i_{k-1} < i_k = n$). As we are on I_0^*, we must have, for all $0 < j < k$, that $r \rightsquigarrow r_{i_j}$. So, we can do the derivation : $B_r \to B_r L_{r_{i_{k-1}}} \to B_r L_{r_{i_{k-2}}} L_{r_{i_{k-1}}} \xrightarrow{*} L_{r_0} \dots L_{r_{i_{k-1}}}$. Then, from the induction hypothesis, we have that, for all $0 < j < k$, $L_{r_{i_j}} \xrightarrow{*} r_{i_j} ... r_{i_{j+1}}$ and so we get the first point of our induction hypothesis.

We now suppose that there is no i such that $a_i = a$. Then, we still have $r \rightsquigarrow r_1$. In addition, due to the tree structure of I_0^*, we must have $r_n = r_1^-$ and $a_1 = a_{n-1}$. We can then apply the induction hypothesis on $r_2 ... r_{n-1}(a_1,a_1)$: if it is true on all database satisfying the unary inclusion dependencies $\mathcal{UID}$ and

containing a tuple $r^-(a_1, c)$, then $B_{r_1^-} \xrightarrow{*} r_2...r_{n-1}$. Finally, we observe that we have the derivation $B_r \xrightarrow{*} L_{r_1} \to r_1 B_{r_1^-} r_1^- \xrightarrow{*} r_1...r_n$ and so we have shown the two points of the inductive claim.

Thus, we have established the forward direction by induction, and it completes the proof of the claimed equivalence. □

Next, to determine in polynomial time whether π_a is equivalent to π_a^{min} (and hence q), we are going to consider all positions where a filter can be added. To do so, we need to define the *root path* of a filter:

Definition C.3 (Root Path). *Let π_a be an execution plan. Let us consider a filter mapping a variable y in the plan to a constant. Then, one can extract a unique path query $r_1...r_n(a, y)$ from the semantics of π_a, starting from the constant a and ending at the variable y. We call this path the* root path of the filter*.*

The existence and uniqueness come from arguments similar to Property 4.2: we can extract a sequence of calls to generate y and then, from the semantics of this sequence of calls, we can extract the root path of the filter. Note that this is different from the definition of the path transformation (Definition B.5): for each call $f(x, y_1, \ldots, y_n)$ with semantics $r_1(x, y_1), \ldots, r_n(y_{n-1}, y_n)$, if y_i is the variable used in the next call or the output variable, then in the root path we only keep the path $r_1 \ldots r_i$, i.e., we do not add $r_{i+1} \ldots r_n r_n^- \ldots r_{i+1}^-$ as we did in Definition B.5.

This definition allows us to characterize in which case the well-filtering plan π_a is equivalent to its minimal filtering plan π_a^{min}, which we state and prove as the following lemma:

Lemma C.4. *Let $q(x) \leftarrow r(a, x)$ be an atomic query, let $\mathcal{UID}$ be a set of UIDs, and let π_a^{min} a minimal filtering plan equivalent to q under $\mathcal{UID}$. Then, for any well-filtering plan π_a defined for q, the plan π_a is equivalent to π_a^{min} iff for each filter, letting $r_1...r_n(a, a)$ be the loop query defined from the root path of this filter, there is a derivation tree such that $B_r \xrightarrow{*} r_1...r_n$ in the grammar $\mathcal{G}_q$.*

It is easy to show the second point of Theorem B.2 once we have the lemma. We have a linear number of filters, and, for each of them, we can determine in PTIME if B_r generates the root path. So, the characterization can be checked in PTIME over all filters, which allows us to know if π_a is equivalent to π_a^{min} in PTIME, as claimed.

Hence, all that remains to do in this appendix section to establish Theorem B.2 is to prove Lemma C.4. We now do so:

Proof. We consider a filter and the root query $r_1...r_n(a, a)$ obtained from its root path.

We first show the forward direction. Let us assume that π_a is equivalent to π_a^{min}. Then, π_a is equivalent to q, meaning that the loop query $r_1...r_n(a, a)$ is true on all database instances satisfying the unary inclusion dependencies

and containing a tuple $r(a, b)$. So, thanks to Property C.2, we conclude that $B_r \xrightarrow{*} r_1...r_n$.

We now show the more challenging backward direction. Assume that, for all loop queries $r_1...r_n(a, a)$ obtained from the loop path of each filter, there is a derivation tree such that $B_r \xrightarrow{*} r_1...r_n$. We must show that π_a^{min} is equivalent to π_a, i.e., it is also equivalent to q. Now, we know that π_a^{min} contains an atom $r(a, x)$ or $r^-(x, a)$, so all results that it returns must be correct. All that we need to show is that it returns all the correct results. It suffices to show this on the canonical database: let I be the instance obtained by chasing the fact $r(a, b)$ by the unary inclusion dependencies. As π_a^{min} is equivalent to q, we know that it returns b, and we must show that π_a also does. We will do this using the observation that all path queries have at most one binding on the canonical database, which follows from Property 1.

Let us call $\pi_a^{\text{no filter}}$ the execution plan obtained by removing all filters from π_a. As we have $B_r \xrightarrow{*} r_1...r_n$ for all root paths, we know from Property C.2 that $r_1...r_n(a, a)$ is true on all databases satisfying the UIDs, and in particular on I. In addition, on I, $r_1...r_n(a, x_1, ...x_n)$ has only one binding, which is the same than $r_1...r_n(a, x_1, ...x_{n-1}, a)$. So, the filters of π_a do not remove any result of π_a on I relative to $\pi_a^{\text{no filter}}$: as the reverse inclusion is obvious, we conclude that π_a is equivalent to $\pi_a^{\text{no filter}}$ on I.

Now, if π_a^{min} contains no filter or contains a filter which was in π_a, we can apply the same reasoning and we get that π_a^{min} is equivalent to $\pi_a^{\text{no filter}}$ on I, and so π_a^{min} and π_a are equivalent in general.

The only remaining case is when π_a^{min} contains a filter which is not in π_a. In this case, we have that the semantics of π_a contains two consecutive atoms $r(a, x)r^-(x, y)$ where one could have filtered on y with a (this is what is done in π_a^{min}). Let us consider the root path of π to y. It is of the form $r_1...r_n(a, a)r(a, x)r^-(x, y)$. We have $B_r \xrightarrow{*} r_1...r_n$ by hypothesis. In addition, as $r \rightsquigarrow r$ trivially, we get $B_r \to B_rL_r \to B_rrr^- \xrightarrow{*} r_1...r_n.r.r^-$. So, $r_1...r_n.r.r^-(a, a)$ is true on I (Property C.2). Using the same reasoning as before, π_a^{min} is equivalent to $\pi_a^{\text{no filter}}$ on I, and so π_a^{min} and π_a are equivalent in general. This concludes the proof.

□

C.2 Proof of Property B.9

We show that we can effectively reverse the path transformation, which will be crucial to our algorithm:

Property B.9. *Given a word w in $\mathcal{R}^*$, a query $q(x) \leftarrow r(a, x)$ and a set of path functions, it is possible to know in polynomial time if there exists a non-redundant minimal filtering execution plan π_a such that $\mathcal{P}(\pi_a) = w$. Moreover, if such a π_a exists, we can compute one in polynomial time, and we can also enumerate all of them (there are only finitely many of them).*

We are going to construct a finite-state transducer that can reverse the path transformation and give us a sequence of calls. To find one witnessing plan, it will suffice to take one run of this transducer and take the corresponding plan, adding a specific filter which we know is correct. If we want all witnessing plans, we can simply take all possible outputs of the transducer.

To construct the transducer, we are going to use the regular expression P_r from Definition 5.2. We know that P_r faithfully represents plans (Theorem B.13), and it is a regular expression. So we will be able to build an automaton from P_r on which we are going to add outputs to represent the plans.

The start node of our transducer is S, and the final node is F. The input alphabet of our transducer is $\mathcal{R}$, the set of relations. The output alphabet is composed of function names f for $f \in \mathcal{F}$, the set of path functions, and of output symbols OUT_i, which represents the used output of a given function. We explain later how to transform an output word into a non-redundant minimal filtering plan.

First, we begin by creating chains of letters from the $w_{f,i}$ defined in Definition 5.2. For a word $w_{f,i} = r_1...r_k$ (which includes the reverse atoms added at the end when $0 \leq i < n$), this chain reads the word $r_1...r_k$ and outputs nothing.

Next, we construct W_0 between two nodes representing the beginning and the end of W_0: S_{W_0} and F_{W_0}. From S_{W_0} we can go to the start of the chain of a final $w_{f,0}$ by reading an epsilon symbol and by outputting the function name f. Then, at the end of the chain of a final $w_{f,0}$, we go to F_{W_0} by reading an epsilon symbol and by outputting a OUT_1 letter.

Similarly, we construct W' between two nodes representing the beginning and the end of W': $S_{W'}$ and $F_{W'}$. From $S_{W'}$ we can go to the beginning of the chain of a final $w_{f,i}$ with $0 < i < n$ (as explained in Definition 5.2) by reading an epsilon symbol and by outputting the function name f. Then, at the end of the chain of a final $w_{f,i}$, we go to $F_{W'}$ by reading an epsilon symbol. The output symbol of the last transition depends on the last letter of $w_{f,i}$: if it is r, then we output OUT_i; otherwise, we output OUT_{i+1}. This difference appears because we want to create a last atom $r(a, x)$ or $r^-(x, a)$, and so our choice of output variable depends on which relation symbol we have.

Last, using the same method again, we construct W between two nodes representing the beginning and the end of W: S_W and F_W. From S_W we can go to the beginning of the chain of a $w_{f,i}$ with $0 < i \leq n$ (as explained in Definition 5.2) by reading an epsilon symbol and by outputting the function name f. Then, at the end of the chain of a final $w_{f,i}$, we go to $F_{W'}$ by reading an epsilon symbol and outputting OUT_i. In this situation, there is no ambiguity on where the output variable is.

Finally, we can link everything together with epsilon transitions that output nothing. We construct W^* thanks to epsilon transitions between S_W and F_W. Then, W^*W' is obtained by linking F_W to $S_{W'}$ with an epsilon transition. We can now construct $P_r = W_0|(W^*W')$ by adding an epsilon transition between S and S_{W_0}, S and S_W, F_{W_0} and F and $F_{W'}$ and F.

We obtain a transducer that we call $\mathcal{T}_{reverse}$.

Let w be a word of $\mathcal{R}^*$. To know if there is a non-redundant minimal filtering execution plan π_a such that $\mathcal{P}(\pi_a) = w$, one must give w as input to $\mathcal{T}_{reverse}$. If there is no output, there is no such plan π_a. Otherwise, $\mathcal{T}_{reverse}$ nondeterministically outputs some words composed of an alternation of function symbols f and output symbols OUT_i. From this representation, one can easily reconstruct the execution plan: The function calls are the f from the output word and the previous OUT symbol gives their input. If there is no previous OUT symbol (i.e., for the first function call), the input is a. If the previous OUT symbol is OUT_k, then the input is the k^{th} output of the previous function. The last OUT symbol gives us the output of the plan. We finally add a filter with a on the constructed plan to get an atom $r(a, x)$ or $r^-(x, a)$ in its semantics in the last possible atom, to obtain a minimal filtering plan. Note that this transformation is related to the one given in the proof of Theorem B.13 in Section C.4, where it is presented in a more detailed way.

Using the same procedure, one can enumerate all possible output words for a given input and then obtain all non-redundant minimal filtering execution plans π_a such that $\mathcal{P}(\pi_a) = w$. We can understand this from the proof of Theorem B.13 in Section C.4, which shows that there is a direct match between the representation of w as words of $w_{f,i}$ and the function calls in the corresponding execution plan. Last, the reason why the set of output words is finite is because the transducer must at least read an input symbol to generate each output symbol.

C.3 Proof of Theorem B.11

In this appendix, we finally show the main theorem of Appendix B:

Theorem B.11. *Given a set of unary inclusion dependencies, a set of path functions, and an atomic query q, the language $\mathcal{L}_q$ captures q.*

Recall that $\mathcal{L}_q$ is the language of the context-free grammar $\mathcal{G}_q$ from Definition 5.1. Our goal is to show that it is a capturing language.

In what follows, we say that two queries are *equivalent* under a set of UIDs if they have the same results on all databases satisfying the UIDs.

Linking $\mathcal{L}_q$ to equivalent rewritings. In this part, we are going to work at the level of the words of $\mathcal{R}^*$ ending by a r or r^- (in the case $q(x) \leftarrow r(a, x)$ is the considered query), where $\mathcal{R}$ is the set of relations. Recall that the full path transformation (Definition B.6) transforms an execution plan into a word of $\mathcal{R}^*$ ending by an atom r or r^-. Our idea is first to define which words of $\mathcal{R}^*$ are interesting and should be considered. In the next part, we are going to work at the level of functions.

For now, we start by defining what we consider to be the "canonical" path query associated to a skeleton. Indeed, from a skeleton in $\mathcal{R}^*$ where $\mathcal{R}$ is the set of relations, it is not clear what is the associated path query (Definition B.4) as there might be filters. So, we define:

Definition C.5 (Minimal Filtering Path Query). *Given an atomic query $q(a, x) \leftarrow r(a, x)$, a set of relations $\mathcal{R}$ and a word $w \in \mathcal{R}^*$ of relation names from $\mathcal{R}$ ending by r or r^-, the **minimal filtering path query** of w for q is the path query of skeleton w taking as input a and having a filter such that its last atom is either $r(a, x)$ or $r^-(x, a)$, where x is the only output variable.*

As an example, consider the word $onAlbum.onAlbum^-.sang^-$. The minimal filtering path query is: $q'(Jailhouse, x) \leftarrow onAlbum(Jailhouse,y)$, $onAlbum^-(y, Jailhouse)$, $sang^-(Jailhouse, x)$, which is an equivalent rewriting of the atomic query $sang^-(Jailhouse, x)$.

We can link the language $\mathcal{L}_q$ of our context-free grammar to the equivalent rewritings by introducing a corollary of Property C.2:

Corollary C.6. *Given an atomic query $q(a, x) \leftarrow r(a, x)$ and a set $\mathcal{UID}$ of UIDs, the minimal filtering path query of any word in $\mathcal{L}_q$ is a equivalent to q. Reciprocally, for any query equivalent to q that could be the minimal filtering path query of a path query of skeleton w ending by r or r^-, we have that $w \in \mathcal{L}_q$.*

Notice that the minimal filtering path query of a word in $\mathcal{L}_q$ is well defined as all the words in this language end by r or r^-.

Proof. We first suppose that we have a word $w \in \mathcal{L}_q$. We want to show that the minimal filtering path query of w is equivalent to q. We remark that the minimal filtering path query contains the atom $r(a, x)$ or $r^-(x, a)$. Hence, the answers of the given query always include the answers of the minimal filtering path query, and we only need to show the converse direction.

Let I be a database instance satisfying the inclusion dependencies $\mathcal{UID}$ and let $r(a, b) \in I$ (we suppose such an atom exists, otherwise the result is vacuous). Let $q'(a, x)$ be the head atom of the minimal filtering path query. It is sufficient to show that $q'(a, b)$ is an answer of the minimal filtering path query to prove the equivalence. We proceed by structural induction. Let $w \in \mathcal{L}_q$. Let us consider a bottom-up construction of the word. The last rule can only be one of the Rule 5.1 or the Rule 5.2. If it is Rule 5.1, then $\exists r_1, \ldots, r_n \in \mathcal{R}$ such that $w = r_1 \ldots r_n r$ and $B_r \xrightarrow{*} r_1 \ldots r_n$. By applying Property C.2, we know that $r_1 \ldots r_n(a, a)$ has an embedding in I. Hence, $q'(a, b)$ is an answer. If the rule is Rule 5.2, then $\exists r_1, \ldots, r_n, \ldots, r_m \in \mathcal{R}$ such that $w = r_1 \ldots r_n r r_{n+1} \ldots r_m r^-$, $B_r \xrightarrow{*} r_1 \ldots r_n$ and $B_{r^-} \xrightarrow{*} r_{n+1} \ldots r_m$. By applying Property C.2 for the two derivations, and remembering that we have $r(a, b)$ and $r^-(b, a)$ in I, we have that $r_1 \ldots r_n(a, a)$ and $r_{n+1} \ldots r_m(b, b)$ have an embedding in I. Hence, also in this case, $q'(a, b)$ is an answer. We conclude that q' is equivalent to q.

Reciprocally, let us suppose that we have a minimal filtering path query of a path query of skeleton w, which is equivalent to q, and that $q'(a, x)$ is its head atom. We can write it either $q'(a, x) \leftarrow r_1(a, x_1), r_2(x_1, x_2), ..., r_n(x_{n-1}, a) r(a, x)$ or $q'(a, x) \leftarrow r_1(a, x_1), r_2(x_1, x_2), ..., r_n(x_{n-1}, x), r^-(x, a)$. In the first case, as q' is equivalent to q, we have $r_1 ... r_n(a, a)$ which is true on all databases I such that I contains a tuple $r(a, b)$. So, according to Property C.2, $B_r \xrightarrow{*} r_1 ... r_n$, and using Rule 5.1, we conclude that $r_1 ... r_n.r$ is in $\mathcal{L}_q$. In the second case, for similar

reasons, we have $B_r \xrightarrow{*} r_1...r_n r^-$. The last r^- was generated by Rule 5.5, using a non-terminal L_r which came from Rule 5.3 using the trivial UID $r \rightsquigarrow r$. So we have $B_r \rightarrow B_r L_r \rightarrow B_r r B_{r^-} r^- \xrightarrow{*} r_1...r_n r^-$. We recognize here Rule 5.2 and so $r_1...r_n r^- \in \mathcal{L}_q$. This shows the second direction of the equivalence, and concludes the proof. □

Linking the path transformation to $\mathcal{L}_q$. In the previous part, we have shown how equivalent queries relate to the context-free grammar $\mathcal{G}_q$ in the case of minimal filtering path queries. We are now going to show how the full path transformation relates to the language $\mathcal{L}_q$ of $\mathcal{G}_q$, and more precisely, we will observe that the path transformation leads to a minimal filtering path query of a word in $\mathcal{L}_q$.

The path transformation operates at the level of the semantics for each function call, transforming the original tree-shaped semantics into a word. What we want to show is that after the path transformation, we obtain a minimal filtering path query equivalent to q iff the original execution path was an equivalent rewriting. To show this, we begin by a lemma:

Lemma C.7. *Let π_a a minimal filtering non-redundant execution plan. The query $\mathcal{P}'(\pi_a)(x)$ is a minimal filtering path query and it ends either by $r(a, x)$ or $r^-(x, a)$, where x was the variable name of the output of π_a.*

Proof. By construction, $\mathcal{P}'(\pi_a)(x)$ is a minimal filtering path query. Let us consider its last atom. In the case where the original filter on the constant a created an atom $r(a, x)$, then the result is clear: an atom $r^-(x, a)$ is added. Otherwise, it means the original filter created an atom $r^-(x, a)$. Therefore, as observed in the last point of the path transformation, the last atom is $r(a, x)$, where we created the new filter on a. □

The property about the preservation of the equivalence is expressed more formally by the following property:

Property C.8. *Let us consider a query $q(x) \leftarrow r(a, x)$, a set of unary inclusion dependencies $\mathcal{UID}$, a set of path functions $\mathcal{F}$ and a minimal filtering non-redundant execution plan π_a constructed on $\mathcal{F}$. Then, π_a is equivalent to q iff the minimal filtering path query $\mathcal{P}'(\pi_a)$ is equivalent to q.*

Proof. First, we notice that we have $\pi_a(I) \subseteq q(I)$ and $\mathcal{P}'(\pi_a)(I) \subseteq q(I)$ as $r(a, x)$ or $r^-(a, x)$ appear in the semantics of $\pi_a(x)$ and in $\mathcal{P}'(\pi_a)(x)$ (see Lemma C.7). So, it is sufficient to prove the property on the canonical database I_0 obtained by chasing the single fact $r(a, b)$ with $\mathcal{UID}$.

We first show the forward direction and suppose that π_a is equivalent to q. Then, its semantics has a single binding on I_0. Let us consider the i^{th} call $c_i(\underline{x_1}, ..., x_j, ...x_n)$ (x_j is the output used as input by another function or the output of the plan) in π_a and its binding: $r_1(y_1, y_2), ..., r_k(y_k, y_{k+1}), ..., r_m(y_m, y_{m+1})$. Then $r_1(y_1, y_2)$, ..., $r_{k-1}(y_{k-1}, y_k), r_k(y_k, y_{k+1})$, ..., $r_m(y_m, y_{m+1}), r_m^-(y_{m+1}, y_m)$, ..., $r_k^-(y_{k+1}, y_k)$ is

a valid binding for the sub-semantics, as the reversed atoms can be matched to the same atoms than those used to match the corresponding forward atoms. Notice that the last variable is unchanged. So, in particular, the variable named x (the output of π_a) has at least a binding in I_0 before at step 3 of the path transformation. In step 4, we have two cases. In the first case, we add $r^-(x, a)$ to the path semantics. Then we still have the same binding for x as π_a is equivalent to q. In the second case, we added a filter in the path semantics, and we still get the same binding. Indeed, by Property 1, b has a single ingoing r-fact in I_0, which is $r^-(b, a)$. This observation means that, in the binding of the path semantics, the penultimate variable was necessary a on I_0. We conclude that $\mathcal{P}'(\pi_a)$ is also equivalent to q.

We now show the backward direction and suppose $\mathcal{P}'(\pi_a)$ is equivalent to q. Let us take the single binding of $\mathcal{P}'(\pi_a)$ on I_0. Let us consider the sub-semantics of a function call $c_i(\underline{x_1}, ..., x_j, ...x_n)$ (where x_j is the output used as input by another function or the output of the plan): $r_1(y_1, y_2)$, ..., $r_{k-1}(y_{k-1}, y'_k), r_k(y'_k, y'_{k+1})$, ..., $r_m(y'_m, y_{m+1}), r^-_m(y_{m+1}, y_m)$, ..., $r^-_k(y_{k+1}, y_k)$. As all path queries have at most one binding on I_0, we necessarily have $y_k = y'_k$, ..., $y_m = y'_m$. Thus the semantics of π_a has a binding on I which uses the same values than the binding of $\mathcal{P}'(\pi_a)$. In particular, the output variables have the same binding. We conclude that π_a is also equivalent to q. □

We can now apply Corollary C.6 on $\mathcal{P}'(\pi_a)$ as it is a minimal filtering path query : $\mathcal{P}'(\pi_a)$ is equivalent to q iff $\mathcal{P}(\pi_a)$ is in $\mathcal{L}_q$. So, π_a is equivalent to q iff $\mathcal{P}(\pi_a)$ is in $\mathcal{L}_q$.

We conclude that $\mathcal{L}_q$ is a capturing language for q. As our grammar $\mathcal{G}_q$ for $\mathcal{L}_q$ can be constructed in PTIME, this concludes the proof of Theorem B.11.

C.4 Proof of Theorem B.13

Theorem B.13. *Let $\mathcal{F}$ be a set of path functions, let $q(x) \leftarrow r(a, x)$ be an atomic query, and define the regular expression P_r as in Definition 5.2. Then the language of P_r faithfully represents plans.*

Proof. We first prove the forward direction: every word w of the language of P_r is achieved as the image by the full path transformation of a minimal filtering non-redundant plan. To show this, let w be a word in the language of P_r. We first suppose we can decompose it into its elements from W and W': $w = w_{f_1,i_1}...w_{f_{n-1},i_{n-1}}.w_{f_n,i_n}$ with w_{f_n,i_n} being final. Let π_a be composed of the successive function calls $f_1, \ldots, f_n$ where the input of f_1 is the constant a and the input of f_k ($k > 1$) is the i^{th}_{k-1} variable of f_{k-1}. For the output variable and the filter, we have two cases:

1. If the last letter of w_{f_n,i_n} is r, the output of the plan is the i^{th}_n variable of f_n and we add a filter to a on the $(i_n + 1)^{th}$ variable.
2. Otherwise, if the last letter of w_{f_n,i_n} is r^-, the output of the plan is the $(i_n + 1)^{th}$ variable of f_n and we add a filter to a on the i^{th}_n variable (except if this is the input, in which case we do nothing).

We notice that π_a is non-redundant. Indeed, by construction, only the first function takes a as input, and all functions have an output used as input in another function. The added filter cannot be on the input of a function as $i_n > 0$. What is more, π_a is also a minimal filtering plan. Indeed, by construction, we create an atom $r(a, x)$ or an atom $r^-(x, a)$ (with x the output of the plan). Let us show that it is the last possible filter. If we created $r^-(x, a)$, it is obvious as x cannot be used after that atom. If we created $r(a, x)$, we know we could not have a following atom $r^-(x, y)$ where one could have filtered on y: this is what is guaranteed by the third point of the definition of the final $w_{f,i}$.

The only remaining point is to show that $\mathcal{P}(\pi_a) = w$. Indeed, for $k < n$, we notice that w_{f_k,i_k} is the skeleton of the sub-semantics of the k^{th} call in π_a. What is less intuitive is what happens for the last function call.

Let us consider the two cases above. In the first one, the output variable is the i_n^{th} variable of f_n. We call it x. The semantics of $\pi_a(x)$ contains $r_{i_n+1}(x, a) = r^-(x, a)$ (as the last letter of w_{f_n,i_n} is r). We are in the second point of step 4 of the path transformation. The skeleton of the end of the path semantics is not modified and it is w_{f_n,i_n}.

In the second case, the output variable is the $(i_n + 1)^{th}$ variable of f_n. We call it x. The semantics of $\pi_a(x)$ contains $r_{i_n+1}(a, x) = r(a, x)$ (as the last letter of w_{f_n,i_n} is r^-). We are in the first point of step 4 of the path transformation. The skeleton of the end of the path semantics is modified to append r^- and it is now $w_{f_n,i_n+1}r = w_{f_n,i_n}$ as expected.

This establishes that $\mathcal{P}(\pi_a) = w$ in the case where w can be decomposed as elements of W and W'. Otherwise, w is in the language of W_0, so $w = w_{f,0}$ where $w_{f,0}$ is final, ends by r^-, thus starts by r. We define π_a as the execution plan composed of one function call f, which takes as input a. The output of the plan is the first output variable of the function. The plan π_a is non-redundant as it contains only one function call. It is also minimal filtering. Indeed, by definition of $w_{f,0}$, the first output variable is on the first atom. So, the semantics of π_a contains an atom $r(a, x)$ where x is the output variable. Besides, it does not contain an atom $r^-(x, y)$ where y is an output of the f by the third point of the definition of a final $w_{f,i}$.

Finally, we have $\mathcal{P}(\pi_a) = w$, and this concludes the first implication. The transformation that we have here is what was performed by the transducer and the method presented in Section C.2. The only difference is that the technique in Section C.2 will consider all possible ways to decompose w into $w_{f,i}$ and into final $w_{f,i}$ to get all possible non-redundant minimal filtering plans.

We now show the converse direction of the claim: the full path transformation maps non-redundant minimal filtering plans to words in the language of P_r. Suppose that we have a non-redundant minimal filtering plan π_a such that $\mathcal{P}(\pi_a) = w$ and let us show that w is in the language of P_r. For all calls which are not the last one, it is clear that the sub-semantics of these calls are the w_{f,i_k} with $i_k > 0$ (as the plan is non-redundant). So the words generated by the calls that are not the last call are words of the language of W^*.

For the last function call, we have several cases to consider.

First, if $\pi_a(x)$ contains a filter, then it means that either the first atom in the semantics of $\pi_a(x)$ is not $r(a,x)$ or, if it is, it is followed by an atom $r^-(x,a)$.

If we are in the situation where the semantics of $\pi_a(x)$ starts by $r(a,x), r^-(x,a)$, then π_a is composed of only one function call f (otherwise, it would be redundant). Then, it is clear that $\mathcal{P}(\pi_a) = w_{f,1}$ with $w_{f,1}$ being final, and we have the correct form.

If we are in the situation where the semantics of $\pi_a(x)$ does not start by $r(a,x)$, we have the two cases (corresponding to the two cases of the forward transformation). We suppose that the last function call is on f, and the output variable is the i^{th} one in f.

If π_a does not contain an atom $r(a,x)$, then it contains an atom $r^-(x,a)$ and the result is clear: the skeleton of the path semantics is not modified and ends by the sub-semantics of f whose skeleton is $w_{f,i}$ and has the correct properties: the last atom is r, the variable after x is not existential (it is used to filter) and the atom after $r(a,x)$ cannot be $r^-(x,y)$ with y an output variable of f as π_a is minimal filtering.

If π_a contains an atom $r(a,x)$, then in the definition of the path transformation, we append an atom $r^-(x,a)$ after the sub-semantics of the f. We then have the path semantics ending by the atom names $w_{f,i}.r^- = w_{f,i-1}$ and $w_{f,i-1}$, which is final, has the adequate properties.

This shows that w is in the language of P_r in the case π_a has a filter because the word generated by the last call is in W'.

Now, we consider the case when π_a does not have a filter. It means that the semantics of π_a starts by $r(a,x)$ and is not followed by an atom $r^-(x,y)$ where y is the output of a function (as π_a is well-filtering). Then, π_a is composed of only one function call f and it is clear that $\mathcal{P}(\pi_a) = w_{f,0}$ which is final. So, in this case, the word w belongs to W_0.

So, w is in the language of P_r in the case π_a does not have a filter. This concludes the proof of the second direction, which establishes the property. □

D Proofs for Section 4 and 5

In this section, we give the missing details for the proof of the claims given in the main text, using the results from the previous appendices. We first cover in Appendix D.1 the missing details of our algorithm. We then show Theorem 4.4 in Appendix D.2, and show Proposition 4.5 in Appendix D.3.

D.1 Details for our algorithm

We now make more precise the last steps of our algorithm, which were left unspecified in the main text:

- Building all possible execution plans $\pi_a(x)$ from a word w of $\mathcal{G}$: this is specifically done by taking all preimages of w by the path transformation, which is done as shown in Property B.9. Note that these are all minimal filtering plans by definition.

– Checking subsets of variables on which to add filters: for each minimal filtering plan, we remove its filter, and then consider all possible subsets of output variables where a filter could be added, so as to obtain a well-filtering plan which is equivalent to the minimal filtering plan that we started with. (As we started with a minimal filtering plan, we know that at least some subset of output variables will give a well-filtering plan, namely, the subset of size 0 of 1 that had filters in the original minimal filtering plan.) The correctness of this step is because we know by Lemma A.2 that non-redundant equivalent plans must be well-filtering, and because we can determine using Theorem B.2 if adding filters to a set of output variables yields a plan which is still an equivalent rewriting.

D.2 Proof of Theorem 4.4

In this appendix, we show our main theorem:

Theorem 4.4. *There is an algorithm which, given an atomic query $q(x) \leftarrow r(a, x)$, a set $\mathcal{F}$ of path function definitions, and a set $\mathcal{UID}$ of UIDs, decides in polynomial time if there exists an equivalent rewriting of q. If so, the algorithm enumerates all the non-redundant plans that are equivalent rewritings of q.*

We start by taking the grammar $\mathcal{G}_q$ with language $\mathcal{L}_q$ used in Theorem B.11 and defined in Definition 5.1 and the regular expression P_r used in Theorem B.13 and defined in Definition 5.2. We make the following easy claim:

Property D.1. *$\mathcal{L}_q \cap P_r$ is a capturing language that faithfully represents plans, and it can be constructed in PTIME.*

Proof. By construction, P_r represents all possible skeletons obtained after a full path transformation (Theorem B.13).

So, as P_r represents all possible execution plans, and as $\mathcal{L}_q$ is a capturing language (proof of Theorem B.11), then $\mathcal{L}_q \cap P_r$ is a capturing language.

The only remaining part is to justify that it can be constructed in PTIME. First, observe that the grammar $\mathcal{G}_q$ for $\mathcal{L}_q$, and the regular expression for P_r, can be computed in PTIME. Now, to argue that we can construct in PTIME a context-free grammar representing their intersection, we will use the results of [18] (in particular, Theorem 7.27 of the second edition). First, we need to convert the context-free grammar $\mathcal{G}_q$ to a push-down automaton accepting by final state, which can be done in PTIME. Then, we turn P_r into a non-deterministic automaton, which is also done is PTIME. Then, we compute a push-down automaton whose language is the intersection between the push-down automaton and the non-deterministic automaton using the method presented in [18]. This method is very similar to the one for intersecting two non-deterministic automata, namely, by building their product automaton. This procedure is done in PTIME. In the end, we obtain a push-down automaton that we need to convert back into a context-free grammar, which can also be done in PTIME. So, in the end, the context-free grammar $\mathcal{G}$ denoting the intersection of $\mathcal{L}_q$ and of the language of P_r can be constructed in PTIME. This concludes the proof. □

So let us now turn back to our algorithm and show the claims. By Property D.1, we can construct a grammar for the language $\mathcal{L}_q \cap P_r$ in PTIME, and we can then check in PTIME if the language of this new context-free grammar is empty. If it is the case, we know that is no equivalent plan. Otherwise, we know there is at least one. We can thus generate a word w of the language of the intersection – note that this word is not necessarily of polynomial-size, so we do not claim that this step runs in PTIME. Now, as P_r faithfully represents plans (Theorem B.13), we deduce that there exists an execution plan π_a such that $\mathcal{P}(\pi_a) = w$, and from Property B.9, we know we can inverse the path transformation in PTIME to gut such a plan.

To get all plans, we enumerate all words of $\mathcal{L}_q \cap P_r$: each of them has at least one equivalent plan in the preimage of the full path transformation, and we know that the path transformation maps every plan to only one word, so we never enumerate any duplicate plans when doing this. Now, by Property B.9, for any word $w \in \mathcal{L}_q \cap P_r$, we can list all its preimages by the full path transformation; and for any such preimage, we can add all possible filters, which is justified by Theorem B.2 and Property C.2. That last observation establishes that our algorithm indeed produces precisely the set of non-redundant plans that are equivalent to the input query under the input unary inclusion dependencies, which allows us to conclude the proof of Theorem 4.4.

D.3 Proof of Proposition 4.5

Proposition 4.5. *Given a set of unary inclusion dependencies, a set of path functions, an atomic query $q(x) \leftarrow r(a, x)$ and a non-redundant execution plan π_a, one can determine in PTIME if π_a is an equivalent rewriting of q.*

First, we check if π_a is well-filtering, which can easily be done in PTIME. If not, using Lemma A.2 we can conclude that π_a is not an equivalent rewriting. Otherwise, we check if π_a is equivalent to its associated minimal filtering plan. This verification is done in PTIME, thanks to Theorem B.2. If not, we know from Theorem B.3 that π_a is not an equivalent rewriting. Otherwise, it is sufficient to show that π_a^{min} is an equivalent rewriting. To do so, we compute $w = \mathcal{P}(\pi_a^{min})$ in PTIME and check if w is a word of the context-free capturing language defined in Theorem B.11. This verification is done in PTIME. By Theorem B.11, we know that w is a word of the language iff π_a is an equivalent rewriting, which concludes the proof.