Equivalent Rewritings on Path Views with Binding Patterns

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.

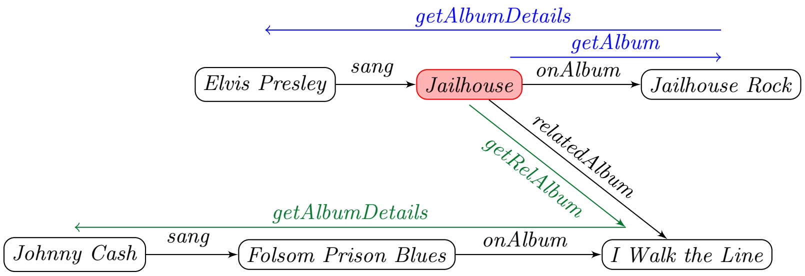


Fig. 1: An equivalent execution plan (blue) and a maximal contained rewriting (green) executed on a database (black).

1 Introduction

In this paper, we study views with binding patterns [25]. 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 paper is complemented by an extended version [27] that contains the proofs for our theorems.

2 Related Work

Formally, we aim at computing *equivalent rewritings* over views with binding patterns [25] 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, 20]. 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 [21] or to complete the set of functions with new functions [22]. 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, 18, 31]. Our approach takes a Local As View perspective, in which all functions are already formulated in the same schema.

Federated Databases. Some works [24, 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) [19], or in Description Logics (DL) [26] 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 [6,9,23] 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 [25], 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:*

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

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 extended version of this paper [27] 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 extended version of this paper [27], 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:*

$$S \rightarrow B_r r \tag{5.1}$$

$$S \rightarrow B_r r B_{r^-} r^- \tag{5.2}$$

$$\forall r_i, r_j \in \mathcal{R} \textit{ s.t. } r_i \rightsquigarrow r_j \textit{ in } \mathcal{UID} : B_{r_i} \rightarrow B_{r_i} L_{r_j} \tag{5.3}$$

$$\forall r_i \in \mathcal{R} : B_{r_i} \rightarrow \epsilon \tag{5.4}$$

$$\forall r_i \in \mathcal{R} : L_{r_i} \rightarrow r_i B_{r_i^-} r_i^- \tag{5.5}$$

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 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 extended version [27]. 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:*

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

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 extended version [27] 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 extended version [27] 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 extended version of this paper [27].

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 [22], 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 [22] 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:

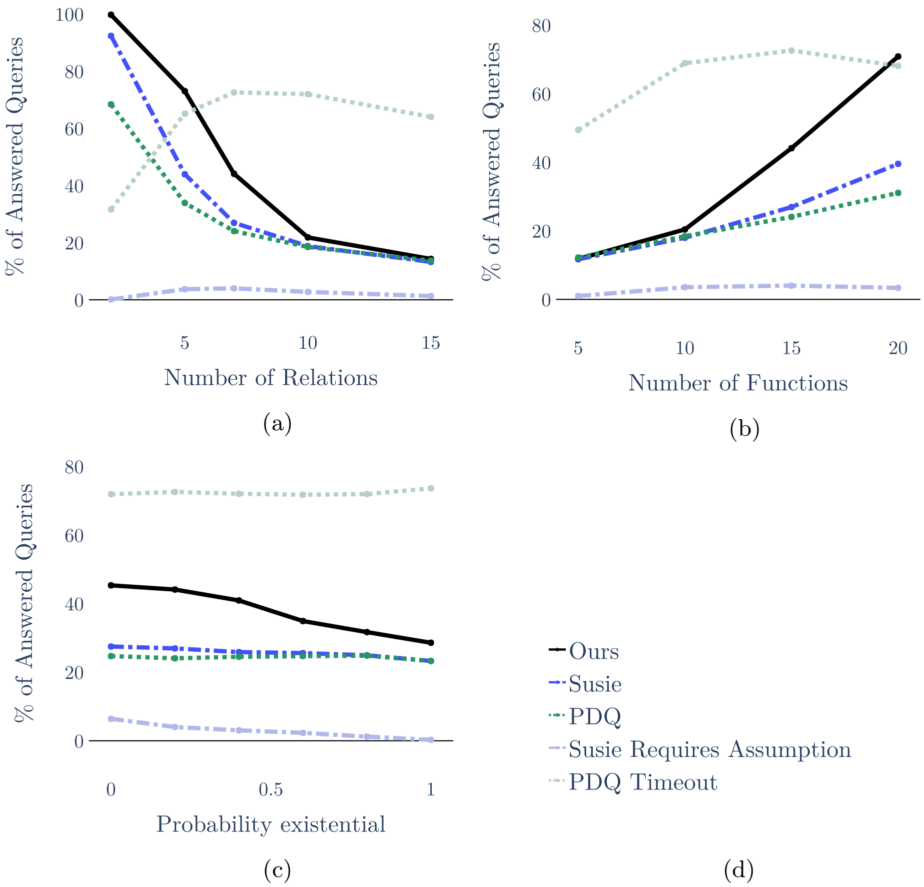
- Ours: The proportion of instances for which our approach found an equivalent plan. As our approach is proved to be correct, this is the true proportion of instances for which an equivalent plan exists.
- Susie: The proportion of instances for which Susie returned a plan which is actually an equivalent 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.



(a)

(b)

(c)

(d)

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 [22], and Movie DB (https://www.themoviedb.org) to replace the (now defunct) Internet Video Archive used in [22]. 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(artistId, collab, collabId) ← hasId^{-}(artistId,artist), isMemberOf(artist,collab), hasId(collab,collabId)
GetBookAuthorAndPrizeByTitle(title, author, prize) ← isTitled^{-}(title, book), wrote^{-}(book,author), hasWonPrize(author,prize)
GetMovieDirectorByTitle(title, 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. Maria Koutraki, Dan Vodislav, and Nicoleta Preda. Deriving intensional descriptions for web services. In *CIKM*, 2015.
19. 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.
20. Alan Nash and Bertram Ludäscher. Processing unions of conjunctive queries with negation under limited access patterns. In *EDBT*, 2004.
21. 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.
22. Nicoleta Preda, Fabian M. Suchanek, Wenjun Yuan, and Gerhard Weikum. SUSIE: Search Using Services and Information Extraction. In *ICDE*, 2013.
23. Ken Q. Pu, Vagelis Hristidis, and Nick Koudas. Syntactic rule based approach to Web service composition. In *ICDE*, 2006.
24. Bastian Quilitz and Ulf Leser. Querying distributed RDF data sources with SPARQL. In *ESWC*, 2008.
25. Anand Rajaraman, Yehoshua Sagiv, and Jeffrey D. Ullman. Answering queries using templates with binding patterns. In *PODS*, 1995.
26. Jinghai Rao, Peep Küngas, and Mihhail Matskin. Logic-based web services composition: From service description to process model. In *ICWS*, 2004.
27. Julien Romero, Nicoleta Preda, Antoine Amarilli, and Fabian Suchanek. Equivalent rewritings on path views with binding patterns, 2020. Extended version with proofs. https://arxiv.org/abs/2003.07316.
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.