Computing and Illustrating Query Rewritings on Path Views with Binding Patterns

Julien Romero
julien.romero@telecom-paris.fr
LTCI, Télécom Paris, Institut Polytechnique de Paris
Palaiseau, France

Nicoleta Preda
nicoleta.preda@uvsq.fr
University of Versailles
Versailles, France

Antoine Amarilli
antoine.amarilli@telecom-paris.fr
LTCI, Télécom Paris, Institut Polytechnique de Paris
Palaiseau, France

Fabian Suchanek
fabian.suchanek@telecom-paris.fr
LTCI, Télécom Paris, Institut Polytechnique de Paris
Palaiseau, France

ABSTRACT

In this system demonstration, we study *views with binding patterns*, which are a formalization of REST Web services. Such views are database queries that can be evaluated using the service, but only if values for the input variables are provided. We investigate how to use such views to answer a complex user query, by rewriting it as an *execution plan*, i.e., an orchestration of calls to the views. In general, it is undecidable to determine whether a given user query can be answered with the available views. In this demo, we illustrate a particular scenario studied in our earlier work [11], where the problem is not only decidable, but has a particularly intuitive graphical solution. Our demo allows users to play with views defined by real Web services, and to animate the construction of execution plans visually.

KEYWORDS

query rewriting; database; visualisation

ACM Reference Format:
Julien Romero, Nicoleta Preda, Antoine Amarilli, and Fabian Suchanek. 2020. Computing and Illustrating Query Rewritings on Path Views with Binding Patterns. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20), October 19–23, 2020, Virtual Event, Ireland*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3340531.3417431

1 INTRODUCTION

Many databases on the Web can be queried through REST Web services: The user accesses a specific parameterised URL, and the service responds by sending back the results of the query. Thus, the parameterised URL acts as a remote function. For example, a music database could offer the function *getAlbum*, which, given a song, returns the album of the song. The user calls this function by accessing a URL like http://music-database.org/getAlbum?song=X, where X is the name of the song. The service then responds by sending back the album of the song X. The advantage of such an interface is that it offers a simple way of accessing the data without downloading all of it. It also allows the data provider to choose which data to expose. According to the Web site programmableweb.com, there are more than 23,000 Web services of this form – including LibraryThing, Amazon, IMDb, Musicbrainz, and Lastfm.

These services typically offer only a limited number of functions. If the query cannot be answered by calling a single function, the user has to combine several function calls. For example, assume that we want to know the singer of the song *Jailhouse* (Figure 1). Assume furthermore that we have two functions at our disposal: *getAlbum* retrieves the album of a song, and *getAlbumDetails* retrieves the songs on the album with their singers. Then we can retrieve the singer of *Jailhouse* by first calling *getAlbum* on *Jailhouse* (retrieving *Jailhouse Rock*), and then *getAlbumDetails* on *Jailhouse Rock* (retrieving all songs on that album with their singers). In this way, we will find the answer to our query, *Elvis Presley*. One can show that under certain conditions [11], this sequence of function calls is a plan that is guaranteed to deliver exactly all answers to the query on all databases: it is an *equivalent rewriting* of the query.

Equivalent rewritings are obviously of primordial interest to the user because they can be used to find the exact answer to the query. Alas, for expressive languages of remote functions, and when assuming arbitrary integrity constraints on the remote data, it is in general undecidable to determine whether a query has an equivalent rewriting on the given set of functions [2, 4]. This is because a rewriting can consist of an arbitrarily complex sequence of function calls. Thus, when we search for a rewriting naively (by enumerating all possible combinations of functions), we do not know when to stop: If we stop, we might miss a rewriting that was just a few steps away. If we do not stop until we found a rewriting, we will keep searching forever if there is none. This is indeed what some state-of-the-art approaches do [2].

We have studied a specific scenario of this problem [11], where functions are paths of binary relations (as in the figure), the query consists of a single relation, and the only permitted constraints are unary inclusion dependencies. Under these conditions, we showed that the problem is not only decidable, but also has a particularly intuitive graphical solution: Equivalent rewritings start from the query constant (*Jailhouse* in Figure 1), walk "forward" to some arbitrary other constants in the database (*Jailhouse Rock* in the

CIKM '20, October 19–23, 2020, Virtual Event, Ireland

ACM ISBN 978-1-4503-6859-9/20/10…$15.00
https://doi.org/10.1145/3340531.3417431

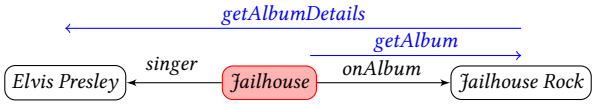

Figure 1: An equivalent rewriting (blue) executed on a database (black).

example), and walk these same paths "backward", finally going back to the initial constant, and into the answer of the query (*Elvis Presley* in the example). Formally, this property can be modelled as an intersection of two formal languages: A context-free language models the type of forward-backward behaviour that we just described, and a regular language models the set of valid plans with the given functions. Since the intersection of these languages is again context-free, it can be determined in polynomial time whether an execution plan exists for a given query.

Thus, the problem has a rather sophisticated formal solution, while at the same time having a rather intuitive graphical interpretation. Our present demo proposal exploits this particularity to make equivalent rewritings understandable through a graphical animation. In this way, our demo playfully introduces the audience to a problem that is both complex and practically relevant.

2 EQUIVALENT REWRITINGS

Let us first introduce the formal side of our problem. In what follows, we assume a vocabulary of binary relation names, such as *singer*, *onAlbum* etc. We also assume that we have, for each relation r, its inverse relation r^-, such that $r(x, y)$ iff $r^-(y, x)$. Our scenario allows for *unary inclusion dependencies* (UIDs) on these relations, i.e., constraints such as the following:

$$\forall x, y : singer(x, y) \Rightarrow \exists z : onAlbum(x, z)$$

This dependency says that if a song x has a singer, then it also has an album.

We consider atomic user queries. These take the form $r(c, x)$, where r is a relation name, c is a constant, and x is a variable. For example, the query *singer(Jailhouse, x)* asks for the singer of the song *Jailhouse*. We model a Web service function as a *view with binding patterns* [2]. In the simple, decidable scenario that we consider [11], such a function takes the following form:

$$f(\underline{x}, x_{i_1}, ..., x_{i_m}) \leftarrow r_1(x, x_1), r_2(x_1, x_2), ..., r_n(x_{n-1}, x_n)$$

Here, f is the function name, x is the input variable, $r_1, ..., r_n$ are relation names, $x_1, ..., x_n$ are the body variables, and $x_{i_1}, \ldots, x_{i_m}$ are the output variables, with $1 \leq i_1 < \cdots < i_m \leq n$. The part to the right of the arrow consists of an ordered sequence of atoms and is called the *body* of the function. A variable that appears in the body, but is neither an input variable nor an output variable is called an *existential variable*. In our example from Figure 1, the functions are:

$$getAlbumDetails(\underline{x}, y, z) \leftarrow onAlbum^-(x, y), singer(y, z)$$
$$getAlbum(\underline{x}, y) \leftarrow onAlbum(x, y)$$

The first function definition says that *getAlbumDetails* requires an input variable x, and delivers two outputs, y and z. These outputs are computed by evaluating the body of the function on the database, i.e., by finding the songs y of the album x, and the singers z of the songs y. An *execution plan* is a chain of function calls. It takes the following form:

$$f_1(\alpha_1, \beta_{1,1}, ...\beta_{1,m_1}), f_2(\alpha_2, \beta_{2,1}, ...\beta_{2,m_2}), ..., f_n(\alpha_n, \beta_{n,1}, ...\beta_{n,m_n})$$

Here, the f_i are function names, α_1 is a constant (e.g., *Jailhouse*), the α_i and $\beta_{k,j}$ are either variables or constants, and if α_i is a variable, it must occur in a preceding function call as a $\beta_{k,j}$. In our example of Figure 1, the plan is:

getAlbum(Jailhouse, y), getAlbumDetails(y, Jailhouse, z)

To execute such a plan, we call the first function, then the second function with each result of the first function, and so on, and return all possible values in the end. From the results of the last call, we select only those where the second element is *Jailhouse*. With this, evaluating the plan is equivalent to executing a large conjunctive query, which consists of the bodies of the functions in the plan with the proper substitutions. In our example, executing the plan is equivalent to running the following query on the database:

onAlbum(Jailhouse, y), onAlbum$^-$*(y,Jailhouse),*
singer(Jailhouse,x) (∗)

Under the UIDs that we assume, this query computes the exact answers of the user query *singer(Jailhouse, x)*. For a full formal definition of functions, queries, unary inclusion dependencies, and execution plans, we refer the reader to [11].

Given a user query, a set of functions, and a (possibly empty) set of UIDs as input, our goal is to find those plans that are *equivalent rewritings* of the query, i.e., that provide the same results as the query on all databases. As we have briefly alluded to, we showed in [11] that these equivalent rewritings have a particular shape: They always consist of a "forward" walk followed by a "backward" walk through the same constants. To make this more precise, let us look at the sequence of relation names in the query (∗) from above:

onAlbum, onAlbum$^-$*, singer*

Here, *onAlbum, onAlbum*$^-$ form a forward-backward path, which is then followed by the relation of the query. More generally, one can show that the sequence of relation names in an equivalent rewriting is always given by a context-free grammar [11], which we will denote G. In its simplest variant[1], the grammar G looks as follows for a query relation q:

$$S \rightarrow (SS \mid rSr^- \mid \epsilon)\, q \quad \text{for all relation names } r$$

For example, if we consider relations a, b, c, d, then G generates (among others) the following sequence:

$$a, b, b^-, c, d, d^-, c^-, a^-, q$$

This moves "forward" (with a, b), then "backward" (with b^-), and again "forward" and "backward" etc., ending by the query relation q.

While the language of G accurately describes the shape of the forward-backward paths, it does not guarantee that these paths can be constructed by a sequence of function calls. In our running example from Figure 1, the path *singer, singer*$^-$ has a forward-backward

[1]The real G is more complex, to account for existential variables. See [11] for details.

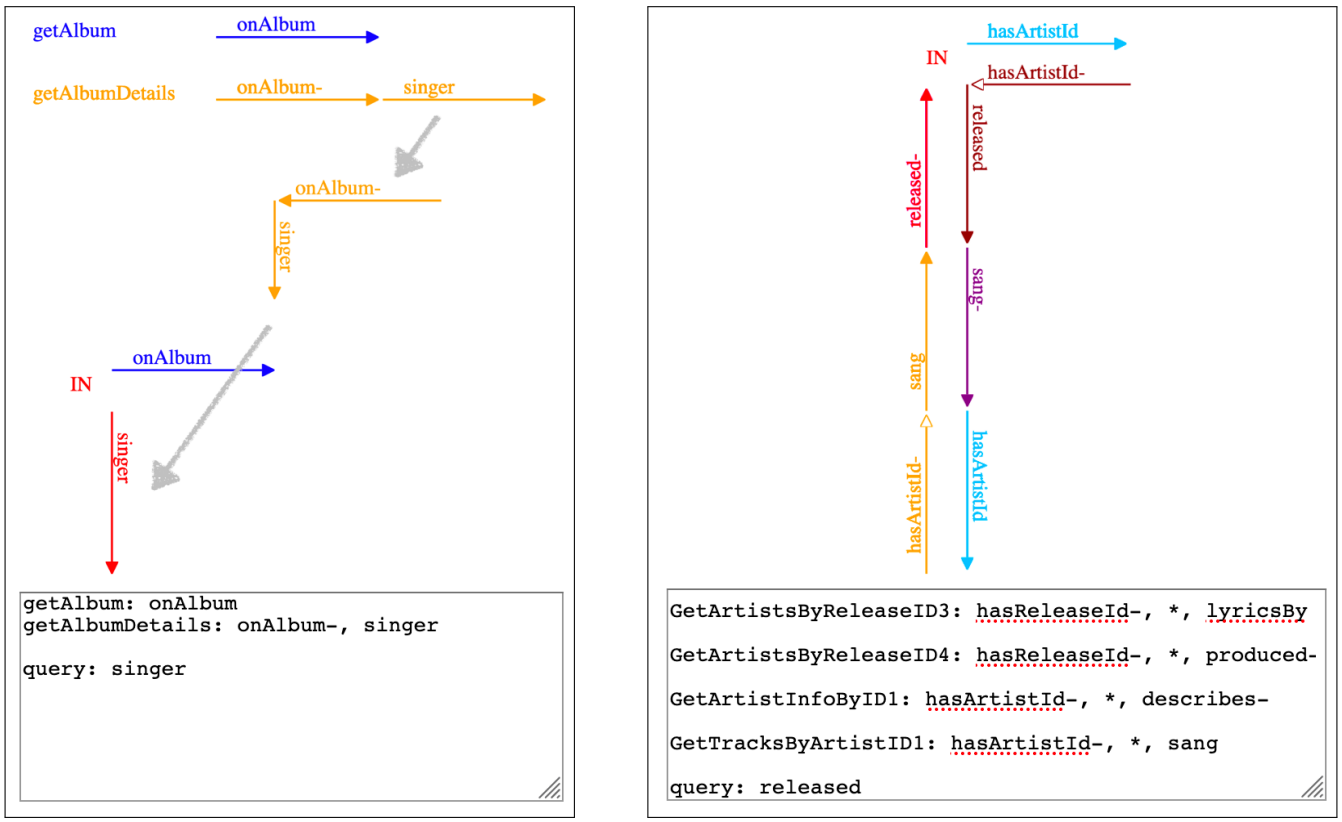

Figure 2: Screenshots of our demo. Left: Our toy example, with the functions on top and the plan being constructed below. The gray arrows indicate the animation. Right: A plan generated for real Web service functions. In both cases, the bottom box contains the function definitions and the query.

shape, but we do not have functions that would allow us to construct a plan of this shape. In order to find those sequences of relations that correspond to execution plans, we need the language of all possible execution plans. In its simplest variant, this language is just the iterated disjunction of the bodies of the functions. In our running example, the language is given by the following regular expression E:

$$(onAlbum \mid onAlbum^{-}\ singer)^{*}$$

This language enumerates the sequences of relations of the queries of execution plans. For example, it enumerates *onAlbum, onAlbum*$^{-}$, *singer*, which corresponds to the plan *getAlbum, getAlbumDetails*. The intersection of the language of E with the language of G is then a context-free language that describes all equivalent rewritings.

Since all equivalent rewritings are, by definition, equivalent to the query, it does not matter which one the user chooses to execute. If the function calls are associated with a cost (network latency, say, or a fee), then the plan with the least number of calls will be best. This is the scenario we adopt for this demo: For a given query, we are only concerned with the shortest plan (breaking ties arbitrarily). Longer plans are usually repetitive and not very instructive.

3 DEMO

While the formal properties of equivalent rewritings are rather sophisticated, our demo gives users a rather playful access to the concept. The user can first specify a set of functions. This is done simply by describing each function as its sequence of relations, as in:

getAlbum : onAlbum
getAlbumDetails : onAlbum^{-}*, singer*

Alternatively, the user can load predefined function definitions of real Web services. We provide function definitions for Abe Books (http://abebooks.com), ISBNDB (http://isbndb.com/), LibraryThing (http://www.librarything.com/), Movie DB (https://www.themoviedb.org), and MusicBrainz (http://musicbrainz.org/) from [9, 11].

The user can then specify a query, by merely giving the relation name. The demo will then search for an execution plan, and animate it to the user. Figure 2 (left) shows how this looks for our running example: The functions are shown on top in blue and orange. The query is shown below as a red edge. The input constant to the query is represented by a generic placeholder "IN". Here, the system has found a plan, and animates it to the user: The blue function was already copied to the right place near the input constant, and the orange function is in the process of moving to the right place (as

illustrated by the grey arrows). When it arrives, it will connect to the first function, and yield the plan shown in Figure 1.

For advanced (or curious) users, our interface shows the grammars of the two languages that were used to compute the plan, as well as the sequence of relations. In this way, the user can see that the path in the animation (here: *onAlbum, onAlbum⁻, singer*) corresponds indeed to a word that is an element of both languages.

Figure 2 (right) shows a more complex example: The user has loaded functions from real Web services – 65 functions in total. The stars in the function definitions (and the corresponding empty arrow heads in their graphical representation) indicate existential variables, i.e., variables that appear in the body of the function, but are not an output variable. The query asks which albums a given artist released. The resulting plan is quite complicated and consists of several forward-backward paths. The animation takes around half a minute: The functions will, one after the other, move into their place to form the final execution plan.

Technically, the user interface of our demo is an HTML page that runs in a Web browser. Behind the scenes, a Flask web server handles the user requests. When the user clicks on the "Find plan" button, the function definitions are sent to a Python implementation of our algorithm from [11] (based on Pyformlang[2]) to compute the plan. The computation takes a few seconds for the most complicated problems. The page then displays the plan as an SVG image, which is animated with `animation` tags. Our server runs on a computer with 2 CPUs of 2.6GHz, and can be easily deployed elsewhere thanks to a Docker container.

4 RELATED WORK

The problem studied in this paper is that of orchestrating views with binding patterns in order to answer user queries. This setting is fundamentally different from answering classical database queries: We cannot execute arbitrary queries. We are limited to executing the views that the services provide. In our running example from Figure 1, we cannot ask directly the query *singer(Jailhouse, x)* (as we could in a classical database setting). Rather, we have to call two functions to obtain the desired answers, and we have to make sure that these functions have their input variables bound before we call them. This is what distinguishes our setting from classical SQL query optimization, from federated databases [10, 12], or from commercial services such as Google BigQuery. These systems can answer arbitrary queries, whereas the services that we discuss in this paper are limited to predefined views. This makes answering queries much harder.

Similar problems to ours have been studied in the context of Web service composition, query rewriting, information integration, execution plan optimization, and reasoning on Web service models. We refer the reader to [11] for an overview and discuss only system demonstrations here. Our work concerns the automatic generation of equivalent rewritings – not actually calling the functions to retrieve results. Other demonstrations call the functions [3, 8], but, unlike our system, they do not always succeed in finding equivalent rewritings for the language that they consider, even when such rewritings exist. Other demonstrations [5] allow exploring federated query execution plans. As we have already seen, these are fundamentally different from our setting. Rule-based semantic specifications for workflows of Web services provided have also been demonstrated in [6]. In our scenario, however, the goal is to find the execution plans in the first place.

Our system uses a wrapper to translate the results of a Web service to RDF. The wrapper can be defined manually, or it can be automatically inferred [7, 13].

Another line of research considers the automated creation of mash-ups [1]. Different from our scenario, the input is not a query, but a partial mashup specification that is to be completed by calls to Web services.

5 CONCLUSION

In this system demonstration, we present a graphical animation of equivalent rewritings for views with binding patterns. Our demo allows users to specify such views (or to load real functions from Web services), and to pose a query. The system will then search for an execution plan, and animate it for the user. It will also show the formal languages used behind the scenes to generate the plan, allowing the user to understand how they are connected to an intuitive graphical interpretation of the plans. As soon as the plan involves several functions, the animation is quite lively, with all the arrows rushing "magically" to their place. Our demo is accessible at http://dangie.r2.enst.fr/ (together with an explanatory video), and the entire code is freely available on Github[3].

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

REFERENCES

[1] Serge Abiteboul, Ohad Greenshpan, Tova Milo, and Neoklis Polyzotis. 2009. MatchUp: Autocompletion for Mashups. In *ICDE, demo track*.
[2] Michael Benedikt, Julien Leblay, Balder ten Cate, and Efthymia Tsamoura. 2016. *Generating Plans from Proofs: The Interpolation-based Approach to Query Reformulation.* Morgan & Claypool Publishers.
[3] Michael Benedikt, Julien Leblay, and Efthymia Tsamoura. 2014. PDQ: Proof-driven query answering over web-based data. *VLDB, demo track* (2014).
[4] Michael Benedikt, Julien Leblay, and Efthymia Tsamoura. 2015. Querying with access patterns and integrity constraints. *VLDB journal* (2015).
[5] Anders Langballe Jakobsen, Gabriela Montoya, and Katja Hose. 2019. How Diverse Are Federated Query Execution Plans Really?. In *ESWC*.
[6] Tobias Käfer, Sebastian Lauber, and Andreas Harth. 2018. Using Workflows to Build Compositions of Read-Write Linked Data APIs on the Web of Things. In *ISWC, demo track*, Marieke van Erp, Medha Atre, Vanessa López, Kavitha Srinivas, and Carolina Fortuna (Eds.).
[7] Maria Koutraki, Dan Vodislav, and Nicoleta Preda. 2015. Deriving Intensional Descriptions for Web Services. In *CIKM*.
[8] Nicoleta Preda, Fabian M. Suchanek, Gjergji Kasneci, Thomas Neumann, Maya Ramanath, and Gerhard Weikum. 2009. ANGIE: Active Knowledge for Interactive Exploration. In *VLDB demo track*.
[9] Nicoleta Preda, Fabian M. Suchanek, Wenjun Yuan, and Gerhard Weikum. 2013. SUSIE: Search Using Services and Information Extraction. In *ICDE*.
[10] Bastian Quilitz and Ulf Leser. 2008. Querying Distributed RDF Data Sources with SPARQL. In *ESWC*.
[11] Julien Romero, Nicoleta Preda, Antoine Amarilli, and Fabian M. Suchanek. 2020. Equivalent Rewritings on Path Views with Binding Patterns. In *ESWC*.
[12] Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. 2011. FedX: Optimization Techniques for Federated Query Processing on Linked Data. In *ISWC*.
[13] Mohsen Taheriyan, Craig A. Knoblock, Pedro A. Szekely, and José Luis Ambite. 2012. Rapidly Integrating Services into the Linked Data Cloud. In *ISWC*.

[2] https://pyformlang.readthedocs.io/

[3] https://github.com/Aunsiels/dangie