

Query Rewriting On Path Views Without Integrity Constraints

Julien Romero, Nicoleta Preda, Fabian Suchanek

▶ **To cite this version:**

Julien Romero, Nicoleta Preda, Fabian Suchanek. Query Rewriting On Path Views Without Integrity Constraints. 2020. hal-03108494

HAL Id: hal-03108494
https://imt.hal.science/hal-03108494

Submitted on 13 Jan 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Query Rewriting On Path Views Without Integrity Constraints

Julien Romero[1][0000−0002−7382−9077], Nicoleta Preda[2], and Fabian Suchanek[1]

[1] LTCI, Télécom Paris, Institut Polytechnique de Paris
{julien.romero,fabian.suchanek}@telecom-paris.fr
[2] University of Versailles nicoleta.preda@uvsq.fr

Abstract. A view with a binding pattern is a parameterised query on a database. Such views are used, e.g., to model Web services. To answer a query on such views, one has to orchestrate the views together in execution plans. The goal is usually to find equivalent rewritings, which deliver precisely the same results as the query on all databases. However, such rewritings are usually possible only in the presence of integrity constraints – and not all databases have such constraints. In this paper, we describe a class of plans that give practical guarantees about their result even if there are no integrity constraints. We provide a characterisation of such plans and a complete and correct algorithm to enumerate them. Finally, we show that our method can find plans on real-world Web Services.

1 Introduction

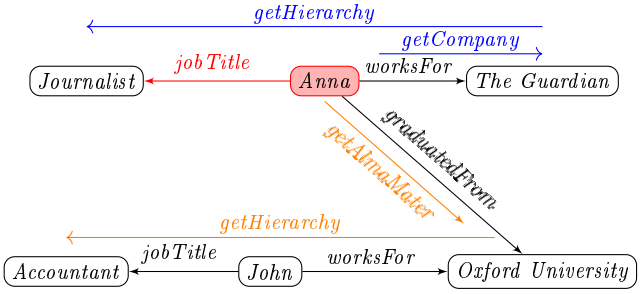


Fig. 1: An equivalent execution plan (blue) and a maximally contained rewriting (orange) executed on a database instance (black).

A view with binding patterns is a parameterised query defined in terms of a global schema [6]. Such a query works like a function: it requires specific values as input and delivers the query results as output. For example, consider the database instance about employees at Figure 1. The call to the

function *getCompany* with an employee *Anna* as input, returns the company *The Guardian* as output. Abstractly, the function is represented as the rule: $getCompany(in, out) \leftarrow worksFor(in, out)$. The *worksFor* relation is of the global schema, which is orthogonal to the schema of the actual data. Unlike query interfaces like SPARQL endpoints, functions prevent arbitrary access to the database engines. In particular, one can model Web forms or REST Web Services as views with binding patterns. According to programmableweb.com, there are currently more than 22,000 such REST Web Services.

If we want to answer a query on a global database that can be accessed only through functions, we have to orchestrate the functions into an execution plan. In our example from Figure 1, if we want to find the job title of Anna, we first have to find her company (by calling *getCompany*), and then her job title (by calling *getHierarchy* on her company, and filtering the results about Anna). Our problem is thus as follows: Given a user query (such as *jobTitle(Anna, x)*) and a set of functions (each being a parameterised conjunctive query), find an execution plan (i.e., a sequence of function calls) that delivers the answer to the query on a database that offers these functions. While the schema of the database is known to the user, she or he does not know whether the database contains the answer to the query at all.

Much of the literature concentrates on finding *equivalent rewritings*, i.e., execution plans that deliver the same result as the original query on all databases that offer this specific set of functions. Unfortunately, our example plan is not an equivalent rewriting: it will deliver no results on databases where (for whatever reasons) Anna has a job title but no employer. The plan is equivalent to the query only if an integrity constraint stipulates that every person with a job title must have an employer and the database instance is complete.

Such constraints are hard to come by in real life, because they may not hold (a person can have a job title but no employer; a person may have a birth date but no death date; some countries do not have a capital[3]; etc.). Even if they hold in real life, they may not hold in the database due to the incompleteness of the data. Hence, they are also challenging to mine automatically. In the absence of constraints, however, an atomic query has an equivalent rewriting only if there is a function that was defined precisely for that query.

This problem appears in particular in data integration settings, where databases are incomplete, and the equivalent rewritings usually fail to deliver results. Therefore, data integration systems often use *maximally contained rewritings* instead of equivalent rewritings. Intuitively speaking, maximally contained rewritings are execution plans that try to find all calls that could potentially lead to an answer. In our example, the plan *getAlmaMater*, *getHierarchy* is included in the maximally contained rewriting: It asks for the university where Anna graduated, and for their job positions. If Anna happens to work at the university where she graduated, this plan will answer the query.

This plan appears somehow less reasonable that our first plan because it works only for people who work at their alma mater. However, both plans are

[3] e.g., the Republic of Nauru

equal concerning their formal guarantees: none of them can guarantee to deliver the answers to the query. This is a conundrum: Unless we have information about the data distribution or more schema information, we have no formal means to give the first plan higher chances of success than the second plan – although the first plan is intuitively much better.

In this paper, we propose a solution to this conundrum: We can show that the first plan (*getCompany*, *getHierarchy*) is "smart", in a sense that we formally define. We can give guarantees about the results of smart plans in the absence of integrity constraints. We also give an algorithm that can enumerate all smart plans for a given **atomic query** and **path-shaped functions** (as in Figure 1). We show that under a condition that we call the *Optional Edge Semantics* our algorithm is complete and correct, i.e., it will exhaustively enumerate all such smart plans. We apply our method to real Web services and show that smart plans work in practice and deliver more query results than competing approaches.

This paper is structured as follows: Section 2 discusses related work, Section 3 introduces preliminaries, and Section 4 gives a definition of smart plans. Section 5 provides a method to characterise smart plans, and Section 6 gives an algorithm that can generate smart plans. We provide extensive experiments on synthetic and real Web services to show the viability of our method in Section 7.

All the proofs and technical details are in the appendix of the accompanying technical report [13].

2 Related Work

Equivalent Rewritings. An equivalent rewriting of a query is an alternative formulation of the query that has the same results as the query on all databases. Equivalent rewritings have also been studied in the context of views with binding patterns [2,12]. However, they may not be sufficient to answer the query [6]. Equivalent rewritings rely on integrity constraints, which may not be available. These constraints are difficult to mine, as most real-life rules have exceptions. Also, equivalent rewritings may falsely return empty answers only because the database instance is incomplete with respect to the integrity constraints. We aim to come up with new relevant rewritings that still offer formal guarantees about their results.

Maximally Contained Rewriting. In data integration applications, where databases are incomplete, and equivalent rewritings are likely to fail, maximally contained rewritings have been proposed as an alternative. A maximally contained rewriting is a query expressed in a chosen language that retrieves the broadest possible set of answers [6]. By definition, the task does not distinguish between intuitively more reasonable rewritings and rewritings that stand little chance to return a result on real databases. For views with binding patterns, the problem has been studied for different rewriting languages and under different constraints [5,4,7]. Some works [8,9] propose to prioritise the execution of the calls in order to produce the first results fast. While the first work [8] does not give guarantees about the plan results, the second one [9] can give guarantees

only for very few plans. Our work is much more general and includes all the plans generated by [9], as we will see.

Plan Execution. Several works study how to optimise given execution plans [14,15]. Our work, in contrast, aims at *finding* such execution plans.

Federated Databases. In federated databases [1,3], a data source supports any queries in a predefined language. In our setting, in contrast, the database can be queried only through *functions*, i.e., specific predefined queries with input parameters.

3 Preliminaries

We use the terminology of [12], and recall the definitions briefly.

Global Schema. We assume a set $\mathcal{C}$ of constants and a set $\mathcal{R}$ of binary relation names. A *fact* $r(a, b)$ is formed from 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.

Queries. An *atom* takes the form $r(\alpha, \beta)$, where $r \in \mathcal{R}$, and α and β are either constants or variables. It can be equivalently written as $r^-(\beta, \alpha)$. 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. A *path query* is a query of the form:

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

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 $1 \leq i \leq n$.

Functions. We model functions as views with binding patterns [10], 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 follow the order of the atoms.

Calling a function for a given value of the input variable means finding the result of the query given by the body of the function on a database instance.

Plans. A *plan* takes the form

$$\pi(x) = c_1, \ldots, c_n, \gamma_1 = \delta_1, \ldots, \gamma_m = \delta_m$$

Here, a is a constant and x is the output variable. Each c_i is a *function call* of the form $f(\underline{\alpha}, \beta_1, \ldots, \beta_n)$, where f is a function name, the input α is either a constant or a variable occurring in some call in $c_1, \ldots, c_{i-1}$, and the outputs $\beta_1, \ldots, \beta_n$ are variables. Each $\gamma_j = \delta_j$ is called a *filter*, where γ_j is an output variable of any call, and δ_j is either a variable that appears in some call or a constant. If the plan has no filters, then we call it *unfiltered*. The *semantics* of the plan is the query

$$q(x) \leftarrow \phi(c_1), \ldots, \phi(c_n), \gamma_1 = \delta_1, \ldots, \gamma_m = \delta_m$$

Here, x is the output variable of the plan, and $\cdot = \cdot$ is an atom that holds in any database instance if and only if its two arguments are identical. 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 given by c_i, and where we have used fresh existential variables across the different $\phi(c_i)$.

To *evaluate* a plan on an instance means running the query above. In practice, this boils down to calling the functions in the order given by the plan. Given an execution plan π_a and a database I, we call $\pi_a(I)$ the answers of the plan on I.

Example 3.1. *Consider our example in Figure 1. There are 3 relation names in the database: worksFor, jobTitle, and graduatedFrom. The functions are:*

$$\begin{aligned} getCompany(\underline{x}, y) &\leftarrow worksFor(\underline{x}, y) \\ getHierarchy(\underline{y}, x, z) &\leftarrow worksFor^-(\underline{y}, x), jobTitle(x, z) \\ getAlmaMater(\underline{x}, y) &\leftarrow graduatedFrom(\underline{x}, y) \end{aligned}$$

The following is an execution plan:

$$\pi_1(z) = getCompany(\underline{Anna}, x), getHierarchy(\underline{x}, y, z), y = Anna$$

The first element is a function call to getCompany with the name of the person (Anna) as input, and the variable x as output. The variable x then serves as input in the second function call to getHierarchy. Figure 1 shows the plan with an example instance. This plan computes the query:

$$worksFor(Anna, x), worksFor^-(x, y), jobTitle(y, z), y = Anna$$

In our example instance, we have the embedding:

$$\sigma = \{x \rightarrow The\ Guardian, y \rightarrow Anna, z \rightarrow Journalist\}.$$

An execution plan π 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.

An *equivalent rewriting* of an atomic query $q(x) \leftarrow r(a, x)$ is an execution plan that has the same results as q on all database instances. For our query language, a *maximally contained rewriting* for the query q is a plan whose semantics contains the atom $r(a, x)$.

4 Defining Smart Plans

Given an atomic query, and given a set of path functions, we want to find a reasonable execution plan that answers the query.

Introductory Observations. Let us consider again the query $q(x) \leftarrow jobTitle(Anna, x)$ and the two plans in Figure 1. The first plan seems to be smarter than the second one. The intuition becomes more formal if we look at the queries in their respective semantics. The first plan is the plan $\pi_1(z)$ given in Example 3.1. Its semantics is the query: $worksFor(Anna, x), worksFor^-(x, y), jobTitle(y, z), y = Anna$. If the first atom has a match in the database instance, then $y = Anna$ is indeed a match, and the plan delivers the answers of the query. If the first atom has no match in the database instance, then the plan returns no result, while the query may have one. To make the plan equivalent to the query on all database instances, we would need the following unary inclusion dependency: $jobTitle(x, y) \rightarrow \exists z : worksAt(x, z)$. In our setting, however, we cannot assume such an integrity constraint. Let us now consider the second plan:

$$\pi_2(z) = getAlmaMater(\underline{Anna}, x), getHierarchy(\underline{x}, y, z), y = Anna$$

Its semantics are: $graduatedFrom(Anna, x), worksFor^-(x, y), jobTitle(y, z), y = Anna$. To guarantee that $y = Anna$ is a match, we need one constraint at the schema level: the inclusion dependency $graduatedFrom(x, y) \rightarrow worksFor(x, y)$. However, this constraint does not hold in the real world, and it is stronger than a unary inclusion dependency (which has an existential variable in the tail). Besides, π_2, similarly to π_1, needs the unary inclusion dependency $jobTitle(x, y) \rightarrow \exists z : graduatedFrom(x, z)$ to be an equivalent rewriting.

Definition. In summary, the first plan, π_1, returns the query answers if all the calls return results. The second plan, π_2, may return query answers, but in most of the cases even if the calls are successful, their results are filtered out by the filter $y = Anna$. This brings us to the following definition of smart plans:

Definition 4.1 (Smart Plan). *Given an atomic query q and a set of functions, a plan π is* smart *if the following holds on all database instances I: If the filter-free version of π has a result on I, then π delivers exactly the answers to the query.*

We also introduce weakly smart plans:

Definition 4.2 (Weakly Smart Plan). *Given an atomic query q and a set of functions, a plan π is* weakly smart *if the following holds on all database instances I where q has at least one result: If the filter-free version of π has a result on I, then π delivers a super-set of the answers to the query.*

Weakly smart plans deliver a superset of the answers of the query, and thus do not actually help in query evaluation. Nevertheless, weakly smart plans can be useful: For example, if a data provider wants to hide private information, like the phone number of a given person, they do not want to leak it in any way, not

even among other results. Thus, they will want to design their functions in such a way that no *weakly smart plan* exists for this specific query.

Every smart plan is also a weakly smart plan. Some queries will admit only weakly smart plans and no smart plans, mainly because the variable that one has to filter on is not an output variable.

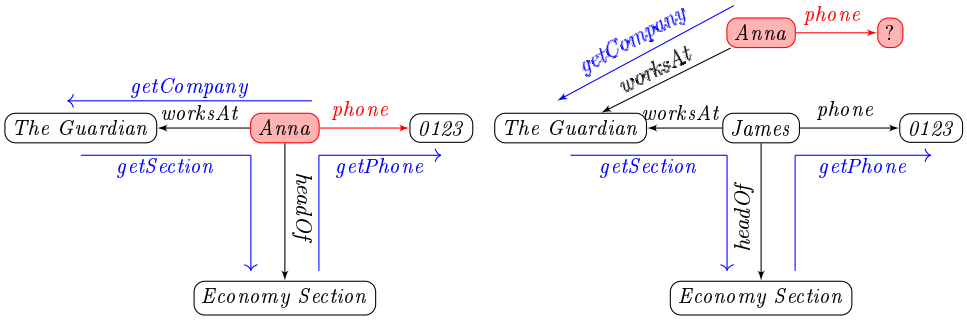


Fig. 2: A non-smart execution plan for the query *phone(Anna,x)*. Left: a database where the plan answers the query. Right: a database where the unfiltered plan has results, but the filtered plan does not answer the query.

Smart plans versus equivalent plans. Consider again the plans π_1 (smart) and π_2 (not-smart) above. Both plans assume the existence of a unary inclusion dependency. The difference is that in addition, π_2 relies on an additional role inclusion constraint. Is it thus sufficient to assume unary inclusion dependencies between all pairs of relations, and apply the algorithm in [12] to find equivalent rewritings? The answer is no: Figure 2 shows a plan that is equivalent if the necessary unary inclusion dependencies hold. However, the plan is not smart. On the database instance shown on the right-hand side, the unfiltered plan returns a non-empty set of results that does not answer the query.

Problem. After having motivated and defined our notion of smart plans, we are now ready to state our problem: Given an atomic query, and a set of path functions, we want to enumerate all smart plans.

5 Characterizing Smart Plans

5.1 Web Service Functions

We now turn to generating smart plans. As previously stated, our approach can find smart plans only under a certain condition. This condition has to do with the way Web services work. Assume that for a given person, a function returns the employer and the address of the working place:

$$getCompanyInfo(\underline{x}, y, z) \leftarrow worksAt(\underline{x}, y), locatedIn(y, z)$$

Now assume that, for some person, the address of the employer is not in the database. In that case, the call will not fail. Rather, it will return only the employer y, and return a null-value for the address z. It is as if the atom *locatedIn(y, z)* were optional. To model this phenomenon, we introduce the notion of *sub-functions*: Given a path function $f : r_1(\underline{x_0}, x_1), r_2(x_1, x_2), \dots r_n(x_{n-1}, x_n)$, the sub-function for an output variable x_i is the function $f_i(\underline{x_0}, ..., x_i) \leftarrow r_1(\underline{x_0}, x_1), \dots r_i(x_{i-1}, x_i)$.

Example 5.1. *The sub-functions of the function getCompanyInfo are* $f_1(\underline{x}, y) \leftarrow worksAt(\underline{x}, y)$*, which is associated to* y*, and* $f_2(\underline{x}, y, z) \leftarrow worksAt(\underline{x}, y), locatedIn(y, z)$*, which is associated to* z*.*

We can now express the Optional Edge Semantics:

Definition 5.2 (Optional Edge Semantics). *We say that we are under the optional edge semantics if, for any path function* f*, a sub-function of* f *has exactly the same binding for its output variables as* f*.*

The optional edge semantics mirrors the way real Web services work. Its main difference to the standard semantics is that it is not possible to use a function to filter out query results. For example, it is not possible to use the function *getCompanyInfo* to retrieve only those people who work at a company. The function will retrieve companies with addresses and companies without addresses, and we can find out the companies without addresses only by skimming through the results after the call. This contrasts with the standard semantics of parametrised queries (as used, e.g., in [12,8,9]), which do not return a result if any of their variables cannot be bound.

This has a very practical consequence: As we shall see, smart plans under the optional edge semantics have a very particular shape.

5.2 Preliminary Definitions

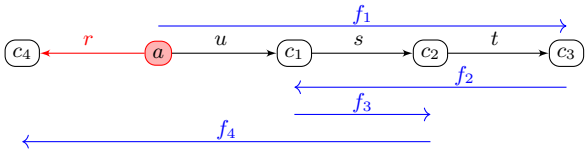


Fig. 3: A bounded plan

Our main intuition is that smart plans under the optional edge semantics walk forward until a turning point. From then on, they "walk back" to the input constant and query (see again Figure 1). As a more complex example, consider the atomic query $q(x) \leftarrow r(a, x)$ and the database shown in Figure 3. The plan f_1, f_2, f_3, f_4 is shown in blue. As we can see, the plan walks "forward" and then

"backward" again. Intuitively, the "forward path" makes sure that certain facts exist in the database (if the facts do not exist, the plan delivers no answer, and is thus trivially smart). If these facts exist, then all functions on the "backward path" are guaranteed to deliver results. Thus, if a has an r-relation, the plan is guaranteed to deliver its object. Let us now make this intuition more formal.

We first observe (and prove in the technical report) that the semantics of any filter-free execution plan can be reduced to a path query. The path query of Figure 3 is:

$$q(a,x) \leftarrow u(a,y_1), s(y_1,y_2), t(y_2,y_3), t^-(y_3,y_2), s^-(y_2,y_1), \\ s(y_1,y_2), s^-(y_2,y_1), u^-(y_1,y_0), r(y_0,x)$$

Now, any filter-free path query can be written unambiguously as the sequence of its relations – the *skeleton*. In the example, the skeleton is:

$$u.s.t.t^-.s^-.s.s^-.u^-.r$$

In particular, the skeleton of an atomic query $q(x) \leftarrow r(a,x)$ is just r. Given a skeleton $r_1.r_2...r_n$, we write $r_1...r_n(a)$ for the set of all answers of the query when a is given as input. For path functions, we write the name of the function as a shorthand for the skeleton of the semantics of the function. For example, in Figure 3, we have $f_1(a) = \{c_3\}$, and $f_1f_2f_3f_4(a) = \{c_4\}$. We now introduce two notions to formalise the "forward and backward" movement:

Definition 5.3 (Forward and Backward Step). *Given a sequence of relations $r_0...r_n$ and a position $0 \leq i \leq n$, a* forward step *consists of the relation r_i, together with the updated position $i+1$. Given position $1 \leq i \leq n+1$, a* backward step *consists of the relation r^-_{i-1}, together with the updated position $i-1$.*

Definition 5.4 (Walk). *A* walk *to a position k ($0 \leq k \leq n$) through a sequence of relations $r_0...r_n$ consists of a sequence of steps (forward or backward) in $r_0...r_n$, so that the first step starts at position $n+1$, every step starts at the updated position of the previous step, and the last step leads to the updated position k.*

If we do not mention k, we consider that $k = 0$, i.e., we cross the sequence of relations entirely.

Example 5.5. *In Figure 3, a possible walk through r^-ust is $t^-s^-ss^-u^-r$. This walk goes from c_3 to c_2 to c_1, then to c_2, and back through c_1, c, c_4 (as indicated by the blue arrows).*

We can now formalise the notion of the forward and backward path:

Definition 5.6 (Bounded plan). *A bounded path for a set of relations $\mathcal{R}$ and a query $q(x) \leftarrow r(a,x)$ is a path query P, followed by a walk through r^-P. A bounded plan for a set of path functions $\mathcal{F}$ is a non-redundant execution plan whose semantics are a bounded path. We call P the forward path and the walk though r^-P the backward path.*

Example 5.7. *In Figure 3, $f_1 f_2 f_3 f_4$ is a bounded path, where the forward path is f_1, and the backward path $f_2 f_3 f_4$ is a walk through $r^- f_1$.*

5.3 Characterising Smart Plans

Our notion of bounded plans is based purely on the notion of skeletons, and does not make use of filters. This is not a problem, because smart plans depend on constraint-free plans. Furthermore, we show in the technical report that we can restrict ourselves to execution plans whose semantics is a path query. This allows for the following theorems (proven in the technical report):

Theorem 5.8 (Correctness)**.** *Let $q(x) \leftarrow r(a, x)$ be an atomic query, F a set of path functions and F_{sub} the set of sub-functions of F. Let π_a be a non-redundant bounded execution plan over the F_{sub} such that its semantics is a path query. Then π_a is weakly smart.*

Theorem 5.9 (Completeness)**.** *Let $q(x) \leftarrow r(a, x)$ be an atomic query, F a set of path functions and F_{sub} the set of sub-functions of F. Let π_a be a weakly smart plan over F_{sub} such that its semantics is a path query. Then π_a is bounded.*

We have thus found a way to recognise weakly smart plans without executing them. Extending this characterisation from weakly smart plans to fully smart plans consists mainly of adding a filter. The technical report gives more technical details.

6 Generating Smart Plans

We have shown that weakly smart plans are precisely the bounded plans. We will now turn to generating such plans. Let us first introduce the notion of minimal plans.

6.1 Minimal Smart Plans

In line with related work [12], we will not generate redundant plans. These contain more function calls, and cannot deliver more results than non-redundant plans. More precisely, we will focus on *minimal plans*:

Definition 6.1 (Minimal Smart Plan). *Let $\pi_a(x)$ be a non-redundant execution plan organised in a sequence $c_0, c_1, \ldots, c_k$ of calls, 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. π_a is a* minimal (weakly) smart plan *if it is a (weakly) smart plan and there exists no other (weakly) smart plan $\pi'_a(x)$ composed of a sub-sequence $c_{i_1}, ..., c_{i_n}$ (with $0 \leq i_1 < ... < i_n \leq k$).*

Example 6.2. *Let us consider the two functions* $f_1(x, y) = r(x, y)$ *and* $f_2(y, z) = r^-(y, t).r(t, z)$. *For the query* $q(x) \leftarrow r(a, x)$, *the plan* $\pi_a(x) = f_1(a, y), f_2(y, x)$ *is obviously weakly smart. It is also non-redundant. However, it is not minimal. This is because* $\pi'_a(x) = f_1(a, x)$ *is also weakly smart, and is composed of a sub-sequence of calls of* π_a.

In general, it is not useful to consider non-minimal plans because they are just longer but cannot yield more results. On the contrary, a non-minimal plan can have fewer results than its minimal version, because the additional calls can filter out results. The notion of minimality would make sense also in the case of equivalent rewritings. However, in that case, the notion would impact just the number of function calls and not the results of the plan, since equivalent rewritings deliver the same results by definition. In the case of smart plans, as we will see, the notion of minimality allows us to consider only a finite number of execution plans and thus to have an algorithm that terminates.

6.2 Bounding and Generating the Weakly Smart Plans

We can enumerate all minimal weakly smart plans because their number is limited. We show in the technical report the following theorem:

Theorem 6.3 (Bound on Plans)**.** *Given a set of relations* $\mathcal{R}$, *a query* $q(x) \leftarrow r(a, x)$, $r \in \mathcal{R}$, *and a set of path function definitions* $\mathcal{F}$, *there can be no more than* $M!$ *minimal smart plans, where* $M = |\mathcal{F}|^{2k}$ *and* k *is the maximal number of atoms in a function. Besides, there exists an algorithm to enumerate all minimal smart plans.*

This bound is very pessimistic: In practice, the plans are very constrained and thus, the complete exploration is quite fast, as we will show in Section 7.

The intuition of the theorem is as follows: Let us consider a bounded path with a forward and a backward path. For each position i, we consider a state that represents the functions crossing the position i (we also consider function starting and ending there). We notice that, as the plan is minimal, there cannot be two functions starting at position i (otherwise the calls between these functions would be useless). This fact limits the size of the state to $2k$ functions (where k is the maximal size of a function, the 2 is due to the existence of both a forward and backward path). Finally, we notice that a state cannot appear at two different positions; otherwise, the plan would not be minimal (all function calls between the repetition are useless). Thus, the algorithm we propose explores the space of states in a finite time, and yields all minimal smart plans. At each step of the search, we explore the adjacent nodes that are consistent with the current state. In practice, these transitions are very constrained, and so the complexity is rarely exponential (as we will see in the experiments).

6.3 Generating the Weakly Smart Plans

Theorem 6.3 allows us to devise an algorithm that enumerates all minimal weakly smart plans. For simplicity, let us first assume that no function definition contains

a loop, i.e., no function contains two consecutive relations of the form rr^-. This means that a function cannot be both on a forward and backward direction. We will later see how to remove this assumption. Algorithm 1 takes as input a query q and a set of function definitions $\mathcal{F}$. It first checks whether the query can be answered trivially by a single function (Line 1). If that is the case, the plan is printed (Line 2). Then, the algorithm sets out to find more complex plans. To avoid exploring states twice, it keeps a history of the explored states in a stack H (Line 3). The algorithm finds all non-trivial functions f that could be used to answer q. These are the functions whose short notation ends in q (Line 4). For each of these functions, the algorithm considers all possible functions f' that could start the plan (Line 5). For this, f' has to be *consistent* with f, i.e., the functions have to share the same relations. The pair of f and f' constitute the first state of the plan. Our algorithm then starts a depth-first search from that first state (Line 6). For this purpose, it calls the function *search* with the current state, the state history, and the set of functions. In the current state, a marker (a star) designates the forward path function.

Algorithm 1: FindMinimalWeakSmartPlans

Data: Query $q(a) \leftarrow r(a,x)$, set of path function definitions and all their sub-functions $\mathcal{F}$
Result: Prints minimal weakly smart plans

```
1 if ∃f = r ∈ F then
2     print(f)
3 H ← Stack()
4 foreach f = r_1...r_n.r ∈ F do
5     foreach f' ∈ F consistent with r_n^-...r_1^- do
6         search({⟨f, n, backward⟩, ⟨f', 1, forward⟩*}, H, F)
```

Algorithm 2 executes a depth-first search on the space of states. It first checks whether the current state has already been explored (Line 1). If that is the case, the method just returns. Otherwise, the algorithm creates the new state S' (Line 3). For this purpose, it considers all positioned functions in the forward direction (Lines 5-7). If any of these functions ends, the end counter is increased (Line 6). Otherwise, we advance the positioned function by one position. The $(*)$ means that if the positioned function happens to be the designated forward path function, then the advanced positioned function has to be marked as such, too. We then apply the procedure to the backwards-pointing functions (Lines 8-11).

Once that is done, there are several cases: If all functions ended, we have a plan (Line 12). In that case, we can stop exploring because no minimal plan can include an existing plan. Next, the algorithm considers the case where one function ended, and one function started (Line 13). If the function that ended were the designated forward path function, then we would have to add one more forward function. However, then the plan would contain two functions that start

at the current state. Since this is not permitted, we just do not do anything (Line 14), and the execution jumps to Line 29. If the function that ended was some other function, then the ending and the starting function can form part of a valid plan. No other function can start or end at the current state, and hence we just move to the next state (Line 15). Next, the algorithm considers the case where one function starts and no function ends (Line 16). In that case, it has to add another backward function. It tries out all functions (Line 17-19) and checks whether adding the function to the current state is consistent (as in Algorithm 1). If that is the case, the algorithm calls itself recursively with the new state (Line 19). Lines 20-23 do the same for a function that ended. Here again, the $(*)$ means that if f was the designated forward path function, then the new function has to be marked as such. Finally, the algorithm considers the case where no function ended, and no function started (Line 24). In that case, we can just move on to the next state (Line 25). We can also add a pair of a starting function and an ending function. Lines 26-28 try out all possible combinations of a starting function and an ending function and call the method recursively. If none of the previous cases applies, then $end > 1$ and $start > 1$. This means that the current plan cannot be minimal. In that case, the method pops the current state from the stack (Line 29) and returns.

Theorem 6.4 (Algorithm)**.** *Algorithm 1 is correct and complete, terminates on all inputs, and runs in time $\mathcal{O}(M!)$, where $M = |\mathcal{F}|^{2k}$ and k is the maximal number of atoms in a function.*

The worst-case runtime of $\mathcal{O}(M!)$ is unlikely to appear in practice. Indeed, the number of possible functions that we can append to the current state in Lines 19, 23, 28 is severely reduced by the constraint that they must coincide on their relations with the functions that are already in the state. In practice, very few functions have this property. Furthermore, we can significantly improve the bound if we are interested in finding only a single weakly smart plan:

Theorem 6.5. *Given an atomic query and a set of path function definitions $\mathcal{F}$, we can find a single weakly smart plan in $\mathcal{O}(|\mathcal{F}|^{2k})$, where k is the maximal number of atoms in a function.*

Functions with loops. If there is a function that contains a loop of the form $r.r^-$, then Algorithm 2 has to be adapted as follows: First, when neither functions are starting nor ending (Lines 24-28), we can also add a function that contains a loop. Let $f = r_1...r_i r_i^-...r_n$ be such a function. Then the first part $r_1...r_i$ becomes the backward path, and the second part $r_i^-...r_n$ becomes the forward path in Line 27.

When a function ends (Lines 20-23), we could also add a function with a loop. Let $f = r_1...r_i r_i^- r_n$ be such a function. The first part $r_1...r_i$ will create a forward state $\langle r_1...r_i, 1, \mathit{forward}\rangle$. The second part, $r_i^-...r_n$ will create the backward state $\langle r_i^-...r_n, |r_1...r_i|, \mathit{backward}\rangle$. The consistency check has to be adapted accordingly. The case when a function starts (Lines 16-19) is handled analogously. Theorems 6.4 and 6.5 remain valid, because the overall number of states is still bounded as before.

Algorithm 2: Search

Data: A state S with a designated forward path function, a set of states $\mathcal{H}$, a set of path functions $\mathcal{F}$
Result: Prints minimal weakly smart plans

if $S \in H$ **then** return
$H.push(S)$
$S' \leftarrow \emptyset$
$end \leftarrow 0$
foreach $\langle r_1...r_n, i, forward\rangle \in S$ **do**
    if $i+1 > n$ **then** $end++$
    else $S' \leftarrow S' \cup \{\langle r_1...r_n, i+1, forward\rangle^{(*)}\}$
$start \leftarrow 0$
foreach $\langle r_1...r_n, i, backward\rangle \in S$ **do**
    if $i = 1$ **then** $start++$
    else $S' \leftarrow S' \cup \{\langle r_1...r_n, i-1, backward\rangle\}$
if $S' = \emptyset$ **then** print(H)
else if $start = 1 \wedge end = 1$ **then**
    if the designated function ended **then** pass
    else search$(S', H, \mathcal{F})$
else if $start = 1 \wedge end = 0$ **then**
    foreach $f \in \mathcal{F}$ **do**
        $S'' \leftarrow S' \cup \{\langle f, |f|, backward\rangle\}$
        if S'' is consistent **then** search$(S'', H, \mathcal{F})$
else if $start = 0 \wedge end = 1$ **then**
    foreach $f \in \mathcal{F}$ **do**
        $S'' \leftarrow S' \cup \{\langle f, 1, forward\rangle^{(*)}\}$
        if S'' is consistent **then** search$(S'', H, \mathcal{F})$
else if $start = 0 \wedge end = 0$ **then**
    search$(S', H, \mathcal{F})$
    foreach $f, f' \in \mathcal{F}$ **do**
        $S'' \leftarrow S' \cup \{\langle f, 1, forward\rangle, \langle f', |f'|, backward\rangle\}$
        if S'' is consistent **then** search$(S'', H, \mathcal{F})$
$H.pop()$

7 Experiments

We have implemented the Susie Algorithm [9] (more details in the technical report), the equivalent rewriting approach [12] (using Pyformlang [11]), as well as our method (Section 6.2) in Python. The code is available on Github[4]. We conduct two series of experiments – one on synthetic data, and one on real Web services. All our experiments are run on a laptop with Linux, 1 CPU with four cores at 2.5GHz, and 16 GB RAM.

[4] https://github.com/Aunsiels/smart_plans

7.1 Synthetic Functions

In our first set of experiments, we use the methodology introduced by [12] to simulate random functions. We consider a set of artificial relations $\mathcal{R} = \{r_1, ..., r_n\}$, and randomly generated path functions up to length 3, where all variables are existential except the last one. Then we try to find a smart plan for each query of the form $q(x) \leftarrow r(a, x), r \in \mathcal{R}$.

In our first experiment, we limit the number of functions to 30 and vary the number n of relations. All the algorithms run in less than 2 seconds in each setting for each query. Figure 4a shows which percentage of the queries the algorithms answer. As expected, when increasing the number of relations, the percentage of answered queries decreases, as it becomes harder to combine functions. The difference between the curve for weakly smart plans and the curve for smart plans shows that it was not always possible to filter the results to get exactly the answer of the query. Weakly smart plans can answer more queries but at the expense of delivering only a super-set of the query answers. In general, we observe that our approach can always answer strictly more queries than Susie and the equivalent rewriting approach.

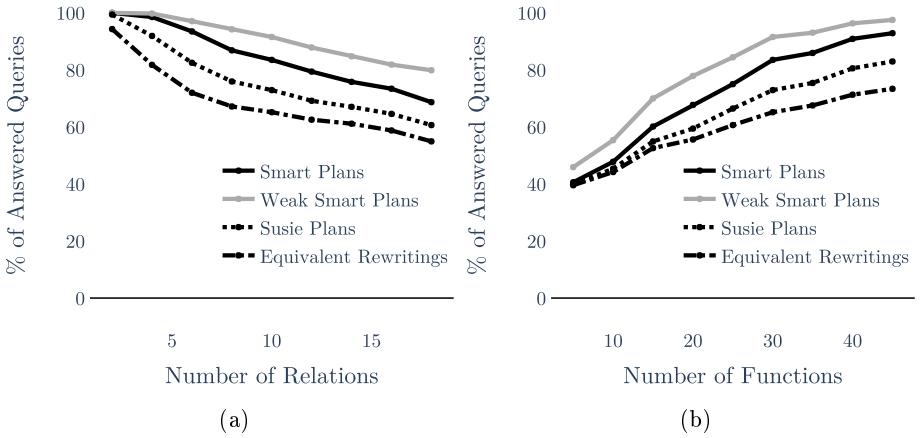


Fig. 4: Percentage of answered queries

In our next experiment, we fix the number of relations to 10 and vary the number of functions. Figure 4b 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 for all approaches. As before, our algorithm outperforms the other methods by a wide margin. The reason is that Susie cannot find all smart plans (see the technical report for more details). Equivalent rewritings, on the other hand, can find only those plans that are equivalent to the query on all databases – which are very few in the absence of constraints.

$getDeathDate(\underline{x}, y, z) \leftarrow hasId^-(x, y) \wedge diedOnDate(y, z)$
$getSinger(\underline{x}, y, z, t) \leftarrow hasRelease^-(x, y) \wedge released^-(y, z) \wedge hasId(z, t)$
$getLanguage(\underline{x}, y, z, t) \leftarrow hasId(x, y) \wedge released(y, z) \wedge language(z, t)$
$getTitles(\underline{x}, y, z, t) \leftarrow hasId^-(x, y) \wedge wrote^-(y, z) \wedge title(z, t)$
$getPublicationDate(\underline{x}, y, z) \leftarrow hasIsbn^-(x, y) \wedge publishedOnDate(y, z)$

Table 1: Examples of real functions (3 of MusicBrainz, 1 of ISBNdb, 1 of LibraryThing).

Web Service	Functions	Relations	Susie	Eq. Rewritings	Smart Plans
MusicBrainz (+IE)	23	42	48% (32%)	48% (32%)	48% (35%)
LastFM (+IE)	17	30	50% (30%)	50% (30%)	50% (32%)
LibraryThing (+IE)	19	32	44% (27%)	44% (27%)	44% (35%)
Abe Books (+IE)	9	8	75% (14%)	63% (11%)	75% (14%)
ISBNdb (+IE)	14	20	65% (23%)	50% (18%)	65% (23%)
Movie DB (+IE)	12	18	56% (19%)	56% (19%)	56% (19%)
UNION with IE	74	82	52%	50%	54%

Table 2: Percentage of queries with smart plans (numbers in parenthesis represent the results with IE).

7.2 Real-World Web Services

In our second series of experiments, we apply the methods to real-world Web services. We use the functions provided by [12,9]. These are the functions of the Web services of Abe Books, ISBNDB, LibraryThing, MusicBrainz, and MovieDB. Besides, as these Web services do not contain many existential variables, we added the set of functions based on information extraction techniques (IE) from [9].

Table 2 shows the number of functions and the number of relations for each Web service. Table 1 gives examples of functions. Some of them are recursive. For example, MusicBrainz allows querying for the albums that are related to a given album. All functions are given in the same schema. Hence, in an additional setting, we consider the union of all functions from all Web services.

Note that our goal is not to call the functions. Instead, our goal is to determine whether a smart plan exists – before any functions have to be called.

For each Web service, we considered all queries of the form $q(x) \leftarrow r(a, x)$ and $q(x) \leftarrow r^-(a, x)$, where r is a relation used in the function definitions of that Web service. We ran the Susie algorithm, the equivalent rewriting algorithm, and our algorithm for each of these queries. The run-time is always less than 2 seconds for each query. Table 2 shows the ratio of queries for which we could find smart plans. We first observe that our approach can always answer at least as many queries as the other approaches can answer. Furthermore, there are cases where our approach can answer strictly more queries than Susie.

The advantage of our algorithm is not that it beats Susie by some percentage points on some Web services. Instead, the crucial advantage

Query	Plan
hasTrackNumber	*getReleaseInfoByTitle, getReleaseInfoById*
hasIdCollaborator	*getArtistInfoByName, getCollaboratorIdbyId,getCollaboratorsById*
publishedByTitle	*getBookInfoByTitle, getBookInfoById*

Table 3: Example Plans (2 of MusicBrainz, 1 of ABEBooks).

of our algorithm is the guarantee that the results are complete. If our algorithm does not find a plan for a given query, it means that there cannot exist a smart plan for that query. Thus, even if Susie and our algorithm can answer the same number of queries on AbeBooks, only our algorithm can guarantee that the other queries cannot be answered at all. Thus, only our algorithm gives a complete description of the possible queries of a Web service.

Rather short execution plans can answer some queries. Table 3 shows a few examples. However, a substantial percentage of queries cannot be answered at all. In MusicBrainz, for example, it is not possible to answer $produced(a, x)$ (i.e., to know which albums a producer produced), $hasChild^{-}(a,x)$ (to know the parents of a person), and $marriedOnDate^{-}(a, x)$ (to know who got married on a given day). These observations show that the Web services maintain control over the data, and do not allow exhaustive requests.

8 Conclusion

In this paper, we have introduced the concept of smart execution plans for Web service functions. These are plans that are guaranteed to deliver the answers to the query if they deliver results at all. We have formalised the notion of smart plans, and we have given a correct and complete algorithm to compute smart plans. Our experiments have demonstrated that our approach can be applied to real-world Web services. All experimental data, as well as all code, is available at the URL given in Section 7. We hope that our work can help Web service providers to design their functions, and users to query the services more efficiently.

References

1. Aebeloe, C., Montoya, G., Hose, K.: A decentralized architecture for sharing and querying semantic data. In: ESWC (2019)
2. Benedikt, M., Leblay, J., ten Cate, B., Tsamoura, E.: Generating Plans from Proofs: The Interpolation-based Approach to Query Reformulation. Synthesis Lectures on Data Management, Morgan & Claypool (2016)
3. Buron, M., Goasdoué, F., Manolescu, I., Mugnier, M.L.: Obi-wan: Ontology-based rdf integration of heterogeneous data. Proc. VLDB Endow. **13**(12), 2933–2936 (Aug 2020). https://doi.org/10.14778/3415478.3415512, `https://doi.org/10.14778/3415478.3415512`
4. Calì, A., Martinenghi, D.: Querying data under access limitations. In: ICDE (2008)
5. Duschka, O.M., Genesereth, M.R.: Answering recursive queries using views. In: PODS (1997)
6. Halevy, A.Y.: Answering queries using views: A survey. In: VLDB J. (2001)
7. Nash, A., Ludäscher, B.: Processing unions of conjunctive queries with negation under limited access patterns. In: EDBT (2004)
8. Preda, N., Kasneci, G., Suchanek, F.M., Neumann, T., Yuan, W., Weikum, G.: Active Knowledge : Dynamically Enriching RDF Knowledge Bases by Web Services. In: SIGMOD (2010)
9. Preda, N., Suchanek, F.M., Yuan, W., Weikum, G.: SUSIE: Search Using Services and Information Extraction. In: ICDE (2013)
10. Rajaraman, A., Sagiv, Y., Ullman, J.D.: Answering queries using templates with binding patterns. In: PODS (1995)
11. Romero, J.: Pyformlang: An educational library for formal language manipulation. In: SIGCSE. Springer International Publishing (2021)
12. Romero, J., Preda, N., Amarilli, A., Suchanek, F.: Equivalent rewritings on path views with binding patterns. In: The Semantic Web. pp. 446–462. Springer International Publishing, Cham (2020), extended version with proofs: `https://arxiv.org/abs/2003.07316`
13. Romero, J., Preda, N., Suchanek, F.: Query rewriting on path views without integrity constraints. In: Datamod (2020), extended version with proofs: `https://arxiv.org/abs/2010.03527`
14. Srivastava, U., Munagala, K., Widom, J., Motwani, R.: Query optimization over web services. In: VLDB (2006)
15. Thakkar, S., Ambite, J.L., Knoblock, C.A.: Composing, optimizing, and executing plans for bioinformatics web services. In: VLDB J. (2005)