

# Active Knowledge: Dynamically Enriching RDF Knowledge Bases by Web Services

## ABSTRACT

The proliferation of knowledge-sharing communities and the advances in information extraction have enabled the construction of large knowledge bases using the RDF data model to represent entities and relationships. However, as the Web and its latently embedded facts evolve, a knowledge base can never be complete and up-to-date. On the other hand, a rapidly increasing suite of Web services provide access to timely and high-quality information, but this is encapsulated by the service interface. We propose to leverage the information that could be dynamically obtained from Web services in order to enrich RDF knowledge bases on the fly whenever the knowledge base does not suffice to answer a user query.

To this end, we develop a sound framework for appropriately generating queries to encapsulated Web services and efficient algorithms for query execution and result integration. The query generator composes sequences of function calls based on the available service interfaces. As Web service calls are expensive, our method aims to minimize the number of calls in order to retrieve results with sufficient recall. Our approach is fully implemented in a complete prototype system. The user can query and browse the RDF knowledge base as if it already contained all the facts from the Web services. This data, however, is gathered and integrated on the fly, transparently to the user. We demonstrate the viability and efficiency of our approach in experiments based on real-life data provided by popular Web services.

## 1. INTRODUCTION

### 1.1 Motivation

Recent projects like DBpedia [4], YAGO[33, 21], Freebase [35], KnowItAll [5], or Intelligence-in-Wikipedia [41] have successfully created very large semantic databases with many millions of facts. The knowledge is typically represented in RDF, the W3C standard for Semantic-Web data. An RDF knowledge base can be seen as a graph, whose nodes are entities (e.g., persons, companies, movies, locations) and whose edges are relationships (e.g., *bornOnDate*, *is-CEOof*, *actedIn*). Often, this graph can be visualized in a browser, so that users can explore the graph interactively. To query the knowledge base, the user (or a program on behalf of the user) can

pose queries in the W3C-endorsed SPARQL[40] language which supports filters and joins in a schema-free manner.

The knowledge stored in these knowledge bases may be huge, but it can never be complete. It inevitably exhibits gaps and these may irritate the user during exploration and knowledge discovery. Consider, for example, a user who is interested in finding more information about Herta Müller, who just received the Nobel prize in literature. Knowledge bases will likely contain incomplete information. For instance, on the day she received the prize, Wikipedia contained her birthdate, birthplace, citizenship, and only a few books and awards, although she is the author of many more novels, stories, essays, and poems. Another query where the knowledge base is bound to be incomplete would be: “Which other 21st-century writers have won prestigious prizes and could be considered for the Nobel prize?”.

On the other hand, there is a growing number of Web services that provide a wealth of high quality information. If these Web services could be tapped for the knowledge graph, many more user queries could be answered. For example, there are several Web services about books and authors (ISBNdb, Amazon, AbeBooks). Other Web services provide data about songs and music albums, movies and videos, etc. The eConsultant<sup>1</sup>, for example, lists hundreds of public Web services, and there is rapid growth. But these services can be accessed only through an encapsulated API; answers to queries are returned in a semi-structured format (XML) but we cannot directly access the data and we cannot observe a database schema that the service may use internally. There are tools for mapping the XML structure of service-call results into the RDF representation that we need for the knowledge graph, but there is no viable solution for automatically generating the service calls that are needed in order to answer a user’s knowledge query with the enhanced recall that we could obtain by tapping on Web services.

A naive solution would exhaustively generate all possible service calls before-hand, and materialize all the data returned by these services and integrate them into the knowledge base. However, this is practically infeasible, due to both query-load constraints and legal reasons, and it could not guarantee the freshness of the knowledge base either. Web sites bound the number of calls coming from the same IP address, or charge each call with a fee. However, if *only* the data that is relevant for a given query could be dynamically retrieved in the current user context, a much larger number of user queries could be answered with satisfactory recall.

This is our vision of *Active Knowledge*. An active knowledge base is a dynamic federation of knowledge sources where some knowledge is maintained locally and other knowledge is dynam-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$10.00.

<sup>1</sup><http://webdeveloper.econsultant.com/web-services-api-services/>

cally retrieved from Web services and mapped into the local knowledge base on the fly. This process should be transparent to the user or application program that runs on the knowledge base, so that the user sees the data from both the local knowledge base and the external Web services as if were a single comprehensive RDF knowledge base. This simplifies the querying and application development against the knowledge base, and would enable a knowledge-as-a-service paradigm as recently pursued by companies like `cyc.com`, `freebase.com`, or `trueknowledge.com` (but none of these is able to tap on third-party Web services). More precisely, the task that we tackle in this paper is, given a SPARQL query, to retrieve the necessary – and only the necessary – data from Web service via automatically generated function calls and dynamically mapping it into the knowledge base. This poses several technical challenges:

(1) Web service calls typically require input parameters of certain types. For example, certain Web services require as input the name of an author, others require the name of a book or an ISBN. The system has to call the appropriate Web services with appropriate input parameters. Even worse, the answer to the query may require the composition of multiple service calls. For example, the Web service of `MusicBrainz` requires as input the *id* of a singer (as defined by the `MusicBrainz` Web site) and returns the titles of the songs of that singer. Hence, if the user starts with the singer's name, the *id* of the singer must be obtained by a prior service call before invoking the request for the song titles.

(2) The data offered by different Web services overlaps, and there are multiple methods from the same Web service API that could fulfill the same purpose. For instance, information about a book can be obtained by a search by author name, by ISBN, by title, or by year. Therefore, different sequences of service calls can be used to retrieve the same data. Given the cost and latency of a Web service call, unnecessary calls have to be avoided by all means. This requires the calls to be planned in a way that redundancy of answers is minimized. Since a Web service API encapsulates the data and does not expose a data source schema, schema properties cannot be exploited for this purpose.

(3) Different Web services come with different quality-of-service properties like latency and coverage. We would like to satisfy the user's information need with a minimal number of calls. Hence, Web service qualities have to be taken into account, too, when computing the best sequence of Web service calls in order to satisfy a user query.

These considerations lead us to the following problem of coupling an RDF knowledge base with external Web services: *Given a SPARQL query against the knowledge base, and given a maximal number of Web service calls that can be executed, compute the largest number of answers to the query.* If infinitely many calls are allowed, compute the maximal number of answers that can be obtained by sequences of Web service calls.

## 1.2 Contributions and Outline

We have developed a system that carries out our paradigm of active knowledge for interactive querying and exploration. Web services act as dynamically and transparently incorporated components of the knowledge base, with seamless on-the-fly integration of service results into query answers.

We propose a Semantic-Web-oriented model for the *declarative definition of service functions*, to register and naturally embed the potentially useful services in the local knowledge base. The Web service results, upon issuing a function call, are transformed into RDF using predefined mappings, and are dynamically added to the local knowledge base during query processing.

The salient property of our system is its reconciliation of two

paradigms: data warehousing and query mediation. We see the warehousing of Web service results as an elegant solution to address the incompleteness of both knowledge bases and Web services. At every moment of the query evaluation, the local knowledge base maintains a measure of the incompleteness of the current answer. Subsequent calls can use the data returned by the previous calls to fill the gaps. No special mechanism is needed to handle the transfer of values from the output of a service call to the input of another call in the compositions of function calls. Despite these innovations, our approach does not require a new query processing engine. Rather, it naturally extends existing work on local RDF query processing, and we actually employ one of the fastest RDF/SPARQL open-source engines for our prototype [26].

This paper's key contributions are: (1) A language to represent Web service interfaces in our framework. The language extends Datalog with limited access patterns [18], by distinguishing between relationships among input parameters that must be verified as a condition to execute the call, and the relationships among output values returned by the call. (2) An algorithm that for automatically generating appropriate service calls, with awareness of call execution costs and quality-of-service properties of the underlying services. The algorithm orders the service calls according to a principled cost model. (3) A prototype system and extensive experiments, using the publicly available YAGO collection [33] as a local knowledge base and additional data dynamically obtained from comprehensive, timely, and popular Web services about books and music.

We discuss related work in Section 2. Sections 3 and 4 describe our Semantic-Web-oriented framework for the dynamic coupling of Web services and the local knowledge base. Section 5 and 6 are dedicated to the query-evaluation algorithms of our framework. Section 7 describes the system architecture of our prototype, and finally, we evaluate our system in Section 8.

## 2. RELATED WORK

A method of a Web Service API is a function that takes as input some parameters and returns as value a semi-structured document, usually XML. Using existent tools [13], mappings can be predefined in the system so that the XML fragments in the results of calls to the function are translated to RDF-style graphs, according to the schema of the knowledge base. Hence, we can simply see a function as a parameterized SPARQL [40] query, whose result is an RDF fragment.

### 2.1 Answering querying using views

As shown in [27], the SPARQL queries can be translated to Datalog queries, hence we are essentially dealing with conjunctive queries on views. Syntactically, the definition of a function is similar to the views with limited access patterns [12]. The functions have limited access patterns because in order to execute a Web service call, one must provide binding values for the input parameters. However, there is a semantic difference between the Web service APIs and the views in data integration systems. The Web sites do not associate to the methods in the Web service APIs a data source schema. Hence, one can not apply optimization techniques e.g., [20], to eliminate redundant views. A site can define two distinct functions with the same signature. For instance, the Web site `last.fm` exports two functions that take as input artist names and return songs sang by that artist. The first function returns the top ten songs, while the second function returns the last ten songs that were listened recently by the users of `last.fm`.

Our problem is similar to that of answering querying using views in data integration system in that we answer user queries by rewrites

ing the initial query into a set of queries (rewritings) that are executed on the remote data sources. However, in our setting the query evaluation is bounded by the maximal number of calls. This changes the approach to solve the problem, as we show next. Depending on the setting (query optimization or data integration) there are two distinct query rewriting problems. In the first case, the views are complete, the number of the views is limited, and the result is an equivalent rewriting. In the second case, the views are incomplete, the algorithm must scale up to a large number of views, and the result is a maximal contained query. Because Web services are incomplete we shall compare only to the second problem. We use as basis of comparison the language of the rewritings.

*Conjunctive queries.* There exist a number of standard algorithms, most notably BUCKET [24] and MINICON [28]. These algorithms, however, do not consider services whose only role is to provide values for the parameters of a subsequent service, even though this might be the only way to use this second service. Furthermore, they do not support recursive query plans. But, as shown in [23], even non recursive queries require recursive query plans if the views have limited access patterns.

*Conjunctive queries with binding patterns.* In [12] (with improvements in [20, 16]), the authors show that for every query there is a finite rewriting using the views, albeit a recursive one. This algorithm proceeds by rewriting a so-called Local As View system into a so-called Global As View system, by inverting each Datalog rule and introducing Skolem terms. The algorithm first computes the consequences of the rules (i.e. it instantiates all possible function calls); then it uses a bottom-up evaluation to compute the answers. The central weak point of this approach for our scenario is the bottom-up computation. It is simply impossible to enumerate all Web service results, because most Web services restrict the number of calls coming from an IP address, so that the algorithm would be stopped before the first step is completed. Our approach, in contrast, searches to minimize the number of calls after which execution answers are output.

The authors of [30, 42, 14] provide a different approach to query rewriting, targeting equivalence of rewritings. This approach, however, assumes that the views are complete.

## 2.2 Knowledge Representation Formalisms

*XML languages.* Since XML is the de facto standard for Web services, one could think of choosing XML as the data model for the global schema. Still, this does not eliminate the necessity of mapping the schema of the Web service to the schema of the knowledge base. With XML as with RDF, the result of a Web service call must be re-structured according to the schema of the knowledge base. Furthermore, it is infeasible to store the semantic graph in XML documents, and to query it using XPath. The query engine would spend most of the time chasing XLink links. Our approach, in contrast, uses the RDF-3X [26] engine, which is a native RDF query engine.

*Object oriented.* Early works in data integration [17] used an object-oriented language for the rewritings. For the same reasons as above, these approaches are less adequate for the Semantic-Web-oriented (RDF-style) knowledge bases we consider.

*Description Logics.* The standard reasoning formalism for RDF data is OWL, with its flavors OWL Full, OWL DL and OWL lite. In [8], it is shown that the Description Logic language *DL-Lite* can be embedded in an extension of the Datalog language. The major extension consists in allowing existentially qualified variables in the heads of the rules. The problem of reasoning on the knowledge base, however, is orthogonal to our problem.

*Sources with querying capabilities.* Recent work has investigated sources that have querying capabilities by themselves [37].

Our work, in contrast, focuses on Web services, which have no such capabilities.

## 2.3 Other styles of data integration

*ActiveXML.* The concept of embedded Web services was introduced by the ActiveXML paradigm [1]. But the ActiveXML framework has no capabilities to compute all the Web call compositions that answer the query. It does not address the problem of answering querying using views.

*Mashup Systems.* In contrast to the mash-up approaches [32, 19], ANGIE acts like a mediator system, where the query dynamically combines data from local and external sources, on demand.

*Linking Open Data Project.* The Linking Open Data Project aims to link Semantic Web resources into a global and distributed graph that spans several Web sites [7]. The resources are linked using URIs and RDF specifications. However, the integration of dynamic components such as Web services has not been considered yet.

## 2.4 Complementary problems

*The Deep Web.* The Deep Web is the part of the Web that is accessible only by Web forms and Web service APIs. There is much work on the automatic construction of wrappers for Web forms (e.g., [9, 15]). This work is complementary to ours, because it is not concerned with the exploitation of Web services. On the contrary, the work can be used to construct Web services from Web forms [31]. These Web services can then provide the input to our system.

*Schema mappings and Data Fusion on the Fly.* In the present work, we assume that the mapping between the schemas of the Web Services and the schema of the knowledge base is given. The (semi) automatic creation of schema mappings has been addressed in a large corpus of works, e.g., [13, 3, 22]. Furthermore, we are not concerned with entity disambiguation and data fusion in this paper. Data fusion is an important component of our system, but it has been addressed in much previous work (e.g., [32]). In this work, we develop a clean model for query rewriting, and we see the disambiguation and data fusion algorithms as an orthogonal problem.

## 2.5 Other Web Services applications

*Web service composition.* A number of works address the problem of automatic composition (or orchestration) of multiple Web services [6, 10, 34, 25]. Our work, in contrast, is concerned with Web services that provide access to external Web databases.

Another problem is to determine the composition of Web services that can answer a parameterized user query [36], or return objects of a given type [29]. In our model, the user queries are not parameterized. Furthermore, in Section 6, we show how to use the constants in the query, as well as the relationships from the knowledge base where the constants appear to reduce the number of Web calls.

# 3. DATA MODEL

## 3.1 RDFS and Semantic Graphs

In tune with recent work [33, 4, 21], we represent our knowledge base in the RDFS standard [38]. In RDFS, knowledge is modeled as a *semantic graph*. A semantic graph is a directed labeled graph, in which the nodes are entities (such as individuals, classes, and literals) and the labeled edges represent relationships between the

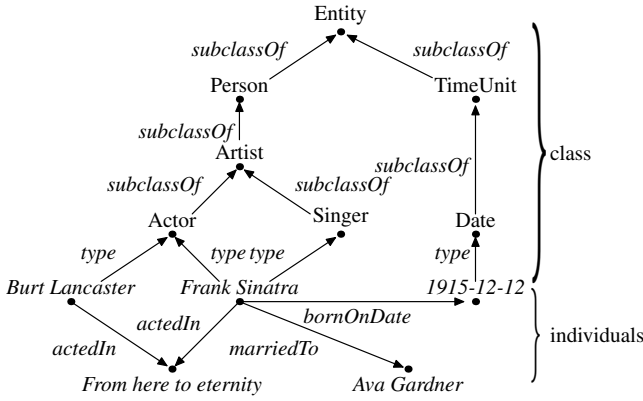


Figure 1: A semantic graph

entities. A fragment of a sample semantic graph is shown in Figure 1. Formally, a semantic graph can be defined as follows.

DEFINITION 1 (SEMANTIC GRAPH). Let  $Rel$  be a set of relation names and let  $Ent \supseteq Rel$  be a set of entities. A semantic graph over  $Rel$  and  $Ent$  is a set of edges  $G \subset Ent \times Rel \times Ent$ .

Thus, a semantic graph is seen as a set of triples. This allows two entities to be connected by two different relationships (e.g., two people can be colleagues and friends at the same time). A triple of a semantic graph is called a *statement*.

In RDFS, there is a distinction between individual entities (such as Frank Sinatra) and class entities (such as the class *singer*). Individuals are linked by the *type* relationship to their class. For example, Frank Sinatra is linked to the class *singer* by an edge (*FrankSinatra, type, singer*). The classes themselves form a hierarchy. More general classes (such as *artist*) include more specific classes (such as *singer*). This hierarchy is expressed in the semantic graph by edges with the *subclassOf* relationship, e.g. (*singer, subclassOf, artist*).

### 3.2 Query Language

As query language, we consider a subset of the standard RDFS query language SPARQL [40].

DEFINITION 2 (QUERY). A query over a set of variables  $Var$  for a semantic graph  $G \subset Ent \times Rel \times Ent$  is a semantic connected graph  $Q \subset (Ent \cup Var) \times (Rel \cup Var) \times (Ent \cup Var)$ .

Figure 2 shows two example queries. The first one asks for the songs sang by Frank Sinatra. The song name is represented by a variable. The second query asks for Frank Sinatra's relationship to Ava Gardner. In this case, the relationship itself is a variable.

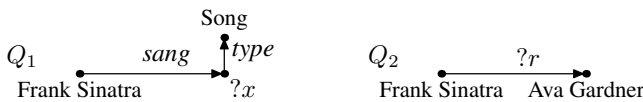


Figure 2: Two sample queries

DEFINITION 3 (QUERY ANSWER). An answer to a query  $Q$  on a semantic graph  $G$  is a graph homomorphism  $\sigma : Q \rightarrow \sigma_Q \subseteq G$  that preserves the entity and relation names, and substitutes the variables in  $Q$  with entities and relationships names from  $G$ .

For instance, consider again the semantic graph shown in Figure 1. The query  $Q_2$  has an answer in the semantic graph because there is the substitution  $\sigma(?r)=marriedTo$ .  $\sigma(Q_2)=(FrankSinatra, marriedTo, AvaGardner)$  is a sub-graph of the semantic graph.

### 3.3 Functions

In our model, the user can query for data that is not yet in the knowledge base. This knowledge is retrieved on the fly by calling Web services. We consider a Web service method as being a function. We see a function as a parameterized query.

DEFINITION 4 (PARAMETERIZED QUERY). A parameterized query is a query  $Q \subset (Ent \cup Var) \times (Rel \cup Var) \times (Ent \cup Var)$ , where the set of variables is partitioned in two sets: input variables, denoted with  $I$ , and output variables, denoted with  $O$ .

The variables in  $I$  must be bound to their actual values before the query is executed. Thus, at execution time, the parameterized query becomes a query as defined above. The answer of the query associates binding values for the variables in  $O$ .

While the local knowledge base is given extensionally, the knowledge provided by the functions is given intensionally. That is, the function definitions do not provide the data itself, but they define a way to obtain it. Conceptually, the extensional data and the intensional data form one large knowledge base. The user can browse this knowledge base transparently, without noticing the difference between intensional and extensional knowledge.

DEFINITION 5 (FUNCTION DEFINITION). A function definition is a parameterized query  $f \subset (Ent \cup I \cup O) \times (Rel \cup I \cup O) \times (Ent \cup I \cup O)$  where the set of edges is partitioned in input edges (input conditions) and output edges so that each variable occurring in an input edge is an input variable.

As a condition to execute the function call, the inputs edge conditions must be verified in the local knowledge. The output edges are instantiated once the result of the function is retrieved.

Consider a function that returns for an author name the books written by that author. The valid inputs are person names. Consequently, a input edge would be the edge ( $?X, type, Person$ ), and as an output conditions the edges ( $?X, type, Author$ ), ( $?X, wrote, ?Y$ ), where  $?Y$  is instantiated by the function call.

In practice, Web services may define also optional input parameters. The optional parameters can be seen as variables having a dual state. They can be either input or output variables. Because a call must provide bindings for at least one input variable, they do not change the problem. With some technical details, the algorithms can handle them. For clarity, we ignore optional parameters here.

**GRAPH REPRESENTATION** In our model, the function definitions themselves are part of the local knowledge base. Every function definition is identified by an entity representing the function. The edges of the function definition are reified. In this process, variable names become nodes in the semantic graph. Each reified edge is connected to the function identifier by an edge representing the *inputEdgeOf* or *outputEdgeOf* relationship. Figure 3 shows an example. The function *getSongs* has the input edge ( $?X, type, Singer$ ). It has as output edge ( $?X, sang, ?Y$ ). To simplify our illustrations, we use continuous lines for reified input edges and dashed lines for reified output edges.

This way, the function definitions are integrated completely into the semantic graph. Thereby, function definitions become first class

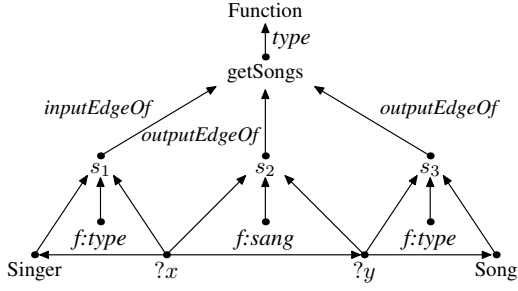


Figure 3: A function definition.

citizens of the knowledge base. Thus, it is possible to query the knowledge base for functions that have certain properties.

**COMPARISON WITH DATALOG** Datalog queries [2] are conjunctive queries of form:

$$q(\bar{X}) \leftarrow r_1(\bar{X}_1), r_2(\bar{X}_2), \dots, r_n(\bar{X}_n)$$

where  $q$  and  $r_1, r_2, \dots, r_n$  are predicate names. The predicate names refer to database relations. The tuples  $\bar{X}_1, \bar{X}_2, \dots, \bar{X}_n$  contain either variables or constants. The query must be *safe*, i.e., that  $\bar{X} \subseteq \bar{X}_1, \bar{X}_2, \dots, \bar{X}_n$  (every variable in the head must also appear in the body).

Furthermore, in order to model the input and the output parameters, adornments attached to queries have been introduced in [30]. If the head of the query has  $n$  attributes, then an adornment consists of a string of length  $n$  composed of the letters  $b$  and  $f$ . The meaning of  $b$  is that a binding value *must* be provided for the variable in that position. For example, the function in Figure 3, can be written in Datalog syntax as follows:

$$\text{getSongs}(?x, ?y)^{bf} \leftarrow \text{sang}(?x, ?y), \text{singer}(?x), \text{album}(?y)$$

where the adornment  $bf$  says that  $?x$  must be bound (input variable) and  $?y$  is free (output variable).

There are two semantic difference between our function definition and the view with limited access pattern [12]. First, the function definition is not a source description, but a description of the result returned by a call to the function. The methods in Web service APIs published by some Web site are not defined with respect to a schema of the source. Hence, one can not apply optimization techniques e.g., [20], to eliminate redundant views. Second, we make the distinction between input and output edges. The role of input edges is to restrict the set of values that are valid inputs for function calls. Previous works ignored that Web sites protect the access to their Web services (or forms) against too many requests coming from the same IP.

**INTENSIONAL RULES** In Datalog, predicates can be defined extensionally (by declaring instantiated atoms) or intensionally (by declaring a rule). This principle carries over naturally to our data model. The role of extensionally defined predicates in Datalog is taken over by concrete edges in the semantic graph. The role of intensionally defined predicates is taken over by *intensional rules*. An intensional rule is given by an ordinary function definition that is not connected to any Web service. Furthermore, all variables that occur in output edges must occur in input edges. If such an intensional rule is called, the output edges are added to the knowledge graph without reference to any Web service. Since the input edges act as conjunctive conditions, intensional rules have the same expressive power as domain-restricted Horn clauses with binary predicates in Datalog.

## 4. QUERY EVALUATION

Given a user query, the problem is to evaluate it using the local knowledge base and the set of functions defined in the system. As described in Section 3.3, we treat the functions as intensional (i.e. active) components of the knowledge base and represent them within the knowledge base. The vision is to unfold (i.e. materialize) intelligently the intensional parts of the knowledge base so that the local knowledge base be able to answer to the user query.

Given a SPARQL query (see Section 3.2), one should compute the sequence of calls whose results, when added to the knowledge base, can answer the query. Note that the functions have input parameters. Hence, for a function, the number of corresponding calls is equal to the cardinality of the function domain. A brute force solution that enumerates and executes all possible function calls is not feasible due to the large volumes of data generated by all the calls.

The problem is challenging because, in order to obtain the desired values for the output of a function, one should not only provide valid input values, but those judicious input values that return the desired result. Furthermore, some input values may only be obtained as the results of other calls. For instance, in Figure 5, the function  $f_1$  can be called to retrieve the books written by *Herta Müller* only if her *id* is provided as input. Input values for the  $f_1$  can be directly obtained from the database, or can be obtained using the function call  $f_2(\text{Herta Müller})$ .

### 4.1 Existing techniques

Algorithms such as BUCKET [24] or MINICON [28] that were developed for data integration systems, can not be directly applied in our framework due to the presence of limited access patterns. These algorithms do not consider functions those only role is to provide values for a second Web call. Closest to our framework, in [12] an algorithm for views with limited access patterns is presented. It is guaranteed to produce the maximal contained rewritings. The focus is on computing the maximal number of answers and not how to compute the first answers fast, with a limited number of calls. The key idea in [12] is to construct a set of inverse rules for each view. An inverse rule shows how to construct tuples for the database relations from the result of a view. For instance, consider a function that given an author name and a conference, returns the papers published by the author in that conference:

$$\text{getPapers}^{bf} (?a, ?p, ?c) \leftarrow \text{authorOf}(?a, ?p), \text{publishedIn}(?p, ?c)$$

For the relations in the body of the rule above, the inverse rules are:

$$\begin{aligned} \text{authorOf}(?a, ?p) &\leftarrow \text{getPapers}^{bf} (?a, ?p, ?c) \\ \text{publishedIn}(?p, ?c) &\leftarrow \text{getPapers}^{bf} (?a, ?p, ?c) \end{aligned}$$

A special rule *dom* (domain) is also added. Its role is to generate the domain where the output variables of functions may take values, union the set of the constants in the query and in the database base. A Datalog query is evaluated on the newly obtained Datalog program using the bottom-up technique [2]. We can not test such technique due to the limitations on the number of Web calls that one IP address can invoke. One can think that the top-down technique [2] can be used instead, because of its properties of pushing the selections. For example, let us consider the query:

$$\text{getPapers}(?p) \leftarrow \text{authorOf}(\text{'Alice'}, ?p), \text{publishedIn}(?p, \text{'SIGMOD'})$$

Then, the term  $\text{authorOf}(\text{'Alice'}, ?p)$  can be expanded in an SLD derivation tree that uses the above defined inverse rules:

$$\begin{aligned} &\text{authorOf}(\text{'Alice'}, ?p), \text{publishedIn}(?p, \text{'SIGMOD'}) \\ &\quad \mid \\ &\text{dom}(?c), \text{getPapers}^{bf}(\text{'Alice'}, ?p, ?c) \end{aligned}$$

Note that the  $?c$  must be bound in order to execute the call. As suggested in [12], the literal  $\text{dom}(?c)$  is inserted in order to be used to bound values for  $?c$ , in a left to right evaluation. Hence, this technique is also limited in its ability to push selections to views with limited access patterns.

Our next algorithms can match at once a subset of edges in a query with a subset of edges in a function. For instance, in our example we can detect that the two edges of the query can be obtained in the answer of the function *getPapers*, and we can directly bind  $?a$  to *Alice* and  $?c$  to *SIGMOD*. We can see this as an extension of the inverse rules. The extension consists in allowing for multiples literals in the heads of the inverse rules.

## 4.2 Preliminaries

Given a query, we denote the list of function calls that are executed in order to evaluate the query, *the evaluation expression*. The execution of each function call in the list comports the following steps: (i) execute the call and retrieve the result, (ii) transform the result to the schema of the local knowledge base using predefined mapping functions (iii) append the transformed result to the knowledge base (iv) compute the new answers generated due to the newly appended results.

A function call is a substitution of the variable in the set  $I$  with actual values from  $\text{Ent} \cup \text{Rel}$ . Before introducing the *function call definition*, let us introduce the concept of *partial function instantiation*. We see a function call as a particular case of a partial instantiation.

**DEFINITION 6 (PARTIAL INSTANTIATION).** A partial instantiation for a function definition  $f \subset (\text{Ent} \cup I \cup O) \times (\text{Rel} \cup I \cup O) \times (\text{Ent} \cup I \cup O)$  is a graph homomorphism

$$\sigma : f \rightarrow J \subset (\text{Ent} \cup \text{Var}) \times (\text{Rel} \cup \text{Var}) \times (\text{Ent} \cup \text{Var})$$

that preserves the names of entities and relationships, and maps variables either to entity and relationship names or to new variables ( $\sigma : I \rightarrow \text{Ent} \cup \text{Var}$  and  $\sigma : O \rightarrow \text{Ent} \cup \text{Var}$ ).

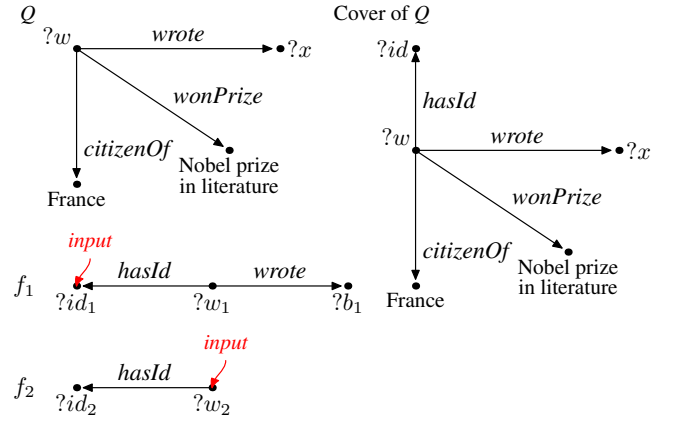
The partial instantiation will instantiate some variables of the function definition with entity or relation names. Other variables of the function definition are simply given new variable names. We denote the elements in the range  $\sigma(I)$  as *input attributes* and the elements in the range  $\sigma(O)$  as *output attributes*.

The problem we have to deal with is the ordering of calls so the results should be obtained after executing a small number of calls. A *function call* for a function  $f$  is a partial instantiation that replaces the variable in  $I$  by entities or relationships from the semantic graph  $G$ , so that the input edges (conditions) are verified in the local knowledge base. The  $O$  variables are mapped to new variables.

**DEFINITION 7 (FUNCTION CALL).** A function call for a function definition  $f \subset (\text{Ent} \cup I \cup O) \times (\text{Rel} \cup I \cup O) \times (\text{Ent} \cup I \cup O)$  is a partial instantiation that maps variables in  $I$  to entity and relationship names so that the required relationships form a sub-graph of the local semantic graph, and maps the variables in  $O$  to new variables in a new variable set  $\text{Var}$ .

$$\sigma : f \rightarrow W \subset (\text{Ent} \cup \text{Var}) \times (\text{Rel} \cup \text{Var}) \times (\text{Ent} \cup \text{Var})$$

The execution of the function call will instantiate the variables in  $\text{Var}$ . The result of a function call  $W$  is a new graph  $R \subset$



**Figure 5: Query  $Q$  and function definitions  $f_1$  and  $f_2$  (left) and query cover  $Q \cup J_{f_1} \cup J_{f_2}$  (right).**

$\text{Ent} \times \text{Rel} \times \text{Ent}$  that is homomorphic with  $W$ . Note that the homomorphism is defined from  $R$  to  $W$  because, the Web sources might be incomplete. Hence, some of the edges in  $W$  might not have correspondence in the result. Finally, we remark that values used as input parameters are always extracted from the local knowledge base (the extensional part of the knowledge base) because the local knowledge base is enriched with the results of previous calls.

**DEFINITION 8 (EVALUATION).** An evaluation for a query  $Q$ , with a set  $F$  of function definitions for Web services, and a semantic knowledge graph  $G \supseteq F$ , is a list of function calls  $W_1, W_2, \dots, W_n$ , with the corresponding results  $R_1, R_2, \dots, R_n$ , so that

$$Q((G \setminus F) \cup R_1 \cup R_2 \dots \cup R_n) \subseteq Q(G)$$

where  $Q(G)$  is the answer of  $Q$  for the knowledge base  $G$ .

If the input value in  $W_j$  is obtained from some value obtained in the result of some call  $W_i$ , then we denote  $W_i \prec W_j$ , and we say that  $W_i$  and  $W_j$  are executed in pipeline.

Note that the query evaluation is based on a dynamic approach. As we shall see in Section 8, the execution of a remote function call is a time consuming operation. This operation is orders of magnitudes slower than a similar local operation. In contrast with the traditional approaches that compute in advance the query evaluation plans, our algorithm tries to minimize the number of calls by using at best the data already gathered locally.

The construction of evaluation expression relies on an intermediary structure, that we define in the following. Because any parameterized query can be wrapped as a function, let  $F_{db}$  denote the set of all queries that can be evaluated on the local knowledge base.

**DEFINITION 9 (QUERY COVER).** A cover for a query  $Q$  on an instance of our data model, is a set of function instantiations  $J_1, J_2, \dots, J_n$  for the functions  $f_1, f_2, \dots, f_n \in F \cup F_{db}$ , on a common graph  $Q \cup J_1 \cup J_2 \cup \dots \cup J_n$ , so that:

- (i) For each input attribute of some  $J_i$ , there is an instantiation  $J_j$  where the attribute is output.
- (ii) For each edge (triple) of  $Q$ , there exist an instantiation  $J_j$  where the edge is output edge.

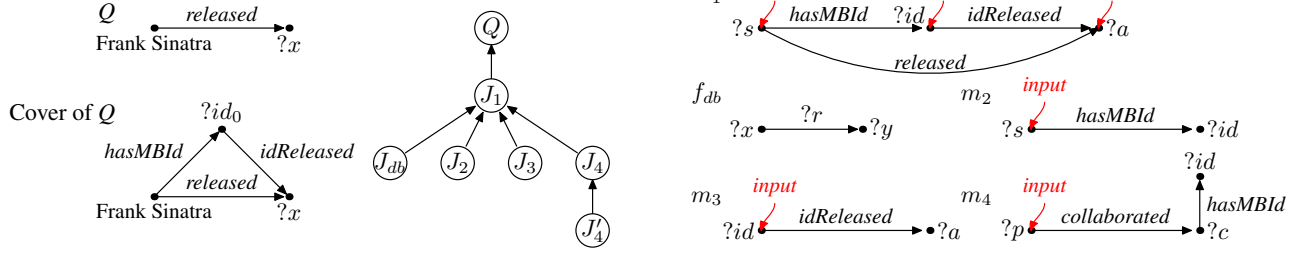


Figure 4: Query and cover(left), derivation tree (center) and function definitions (right).

For two partial instantiation  $J_i$  and  $J_j$  that have the property (i) above, we denote  $J_j \prec J_i$ . Note that two Web calls  $W_j \prec W_i$  correspond to two instantiation  $J_i$  respective  $J_j$  so that  $J_j \prec J_i$ . One can see the partial instantiation as nodes in a directed graph structure where the edges are  $\prec$  relationships.

EXAMPLE 1. Consider the Figure 5. More RDF triples matching the edge  $(?w, \text{wrote}, ?x)$  can be obtained as the result of calls to the function  $f_1$ . A partial instantiation  $J_{f_1}$  for  $f_1$  is then defined so that  $\sigma_1(?id_1) = ?id$  and  $\sigma_1(?w_1) = ?w$ . The graph in the left side represents the query cover. Now,  $f_2$  can provide triples matching the edge  $(?w, \text{hasId}, ?id)$ . We add to the query cover  $J_{f_2}$ , a partial instantiation for  $f_2$ , so that  $\sigma_2(?id_2) = ?id$  and  $\sigma_2(?w_2) = ?w$ . Note that  $?id$  is output attribute in  $J_{f_2}$  and input attribute in  $J_{f_1}$ . Hence  $J_{f_2} \prec J_{f_1} \prec Q$ . Furthermore, the function  $f_3$  can provide relationships that match the edge  $(?w \text{ wonPrize } ?p)$ . In summary, we have three partial instantiations in the query cover.

$$\begin{aligned} J_{f_1} &= \{(?w, \text{wrote}, ?x), (?w, \text{hasId}, ?id)\} \\ J_{f_2} &= \{(?w, \text{hasId}, ?id)\} \\ J_{f_3} &= \{(?w, \text{wonPrize}, \text{"Nobel prize in literature"})\} \end{aligned}$$

EVALUATION FOR INFINITE NUMBER OF WEB CALLS In [23], the authors show that for views with limited access patterns there might be no bound on the size of the rewriting. As an example, consider the following function:

$$f^{bf}(?x, ?y) \leftarrow \text{Artist}(?x), \text{Artist}(?y), \text{Collaborated}(?x, ?y)$$

and a query requesting all the entities of type *Artist*. For each artist in the knowledge base and, for each retrieved artist, a lot more artists could be discovered by calling the function  $f$ . Note that the chain of recursive calls of  $f$  is not bound by the size of the query. In general, one can envision Web services which implement functions that generate infinite domains. For instance, consider a function that given a year, returns the next year in which a total sun eclipse will occur.

## 5. DEPTH-FIRST ALGORITHM

This algorithm relies on a depth-first-search strategy to expand the query cover and on a left to right evaluation. Algorithm 1 computes and outputs the rewritings for a query in the spirit of the chronological backtracking strategy used in Prolog.

The algorithm implements a recursive depth-first search for constructing the derivation tree. The query cover is constructed by recursive calls to the function DF. The function takes as input the set of functions  $F$ , the current configuration of the query cover  $C$ , a stack  $St$  with the instantiations  $J$  for which the derivation tree

---

**Algorithm 1** DF(Funct  $F$ , Cover  $C$ , Stack  $St$ , Query  $Q$ )

---

```

1:  $J \leftarrow St.pop()$ ;
2: if  $(J.depth = MAX \parallel St.empty)$  then
3:   return  $C$ ;
4: end if
5: for  $(f \in \{f_{db}\} \cup F)$  do
6:   for  $(J_i \text{ so that } (J_i = \sigma(f)) \wedge (J_i \prec J) \wedge (J_i \notin C))$  do
7:      $J.addChild(J_i)$ ;
8:     executeWebCallsFor( $J_i$ )
9:     DF( $C \cup J_i, St.push(J_i)$ );
10:   end for
11: end for

```

---

should be computed. The function  $f_{db}$  is a local function that is added for uniformity, so that both the extensional predicates and the intensional predicates are accessed using functions. In the beginning, both the stack  $St$  and the query cover consist of the query  $Q$ . At each recursion step, one partial instantiation  $J$  is removed from  $St$  and new partial instantiations  $J_i$  so that  $J_i \prec J$  are added to  $C$  and to the stack  $St$ .

Consider for instance the query  $Q$  in Figure 4, and the function definitions defined on the left-hand side of the figure.  $Q$  asks for the albums released by Frank Sinatra,  $m_1$  is a function-composition rule equivalent to the Horn clause:

$$\text{released}(?s, ?a) \leftarrow \text{hasMBID}(?s, ?id), \text{isReleased}(?id, ?a)$$

function  $m_2$  maps an artist to its *id* in the MusicBrainz world,  $m_3$  maps the *id* to the albums released by the artist and  $m_4$  returns the collaborators and the *id* of the collaborators.

The algorithm searches a substitution that satisfies the edge  $q$ . In our case,  $m_1$  can satisfy  $q$  with  $\sigma(?s) = \text{Frank Sinatra}$  and  $\sigma(?a) = ?x$ . Hence, the partial instantiation  $J_1 = \sigma(m_1)$  is appended to the query cover, and  $J_1$  is pushed into the stack. In the derivation tree we have  $J_1 \prec Q$ . In the next steps, the algorithm removes  $J_1$  from the stack and unfolds the derivation tree for it. It tries to find the function that satisfy the edges marked with *hasMBID* and *idReleased*. The algorithm always tries to use the local database function to early bind input parameters. In our example, the algorithm chooses  $f_{db}$  to satisfy the edge marked with *hasMBID*. The *id* of *Frank Sinatra* is extracted from the database. Note that another possible choice is  $m_2$ . In the next step (if at least one of the function returned an *id* of *Frank Sinatra*), the algorithm chooses  $m_3$  to satisfy the edge marked with *idReleased*. The result is a rewriting that “covers” the initial query completely. The tree in Figure 4 denotes the derivation tree that is constructed by the DF algorithm.  $J_1, J_2$ , respective  $J_3$  denote the partial instantiations for  $m_1, m_2$ , respective  $m_3$ .

Note that the presented algorithm relies excessively on a depth-first search strategy. In case there is an infinite rewriting of the input query, the algorithm will descend into a non-terminating recursion. For instance, note that the function  $m_4$  returns also relationships for the relation *hasMBId*. Furthermore, the function can be called recursively, so that more *hasMBId* are returned. For the completeness of the solution, one should take into account these relationships. In order to prevent infinite loops, we bound the depth of the derivation tree by *MAX*. Our next algorithm exploits an intuitive cost model for the early binding input parameters in function instantiations, and to prioritize the evaluation of partial instantiations.

## 6. F-RDF ALGORITHM

For a function  $f$ , one can extract from the database the values that can be used as input bindings for  $f$ , and execute the set of function calls. Note that the sequence of calls might be unbound, because the response of a call can contain new binding values for the inputs for  $f$ . We call the list of all function calls that can be defined for  $f$ , the *exhaustive list of calls for  $f$* . This algorithm avoids producing the exhaustive list of calls from the beginning, and instead, searches first for a better strategy. The key is to choose first those bindings that verify some of the conditions in the query.

### 6.1 Lazy construction of the query cover

To simplify, but without losing the generality of the solution, we ignore in the following the typing edges in the query cover  $Q \cup J_1 \dots \cup J_n$ . For each partial instantiation  $J$  in the cover, we construct local queries that extract bindings for the input attributes. The queries are constructed as connected sub-graphs of the query cover. The local queries contain at least a node that is labeled with a constant in the initial query. We denote the local queries using the term *binding queries*.

For example, in the query cover in Figure 5, the Web calls corresponding to  $J_{f_1} = \sigma_1(f_1)$  provide triples that match the edge  $(?w, wrote, ?x)$ . Note that the input attribute  $?id$  must be bound in order to execute a Web call. The node  $?id$  is linked to the node  $?w$  from  $Q$  via the edge  $(?w, hasId, ?id)$ . Now, one should select bindings for  $?w$ . Those values that verify already some constraints in  $Q$  are better candidates to be part of the final solution. Several subqueries of  $Q$  can be considered. For instance, one can select values for  $?w$  so that:

$$\{ (?w, citizenOf, 'France'), (?w, citizenOf, 'Nobel in literature') \}$$

Let  $B$  be a binding query for a partial instantiation  $J$ . For each binding query  $B$ , we can write it as the conjunction of two queries  $B = Q_B \cup P_B$ , where  $Q_B \subseteq Q$ .  $P_B$  is a path (or union of paths) from an input attribute in  $J$  to a node in  $Q$ . We construct the binding queries, incrementally, as follows.

*Case (1)* If  $J \cap Q \neq \emptyset$ . Because the query cover is connected (union of connected graphs), then for each input attribute in  $J$ , there is a path to nodes in  $Q$ .

*Case (2)* If  $J \cap Q = \emptyset$ , then there is a sequence

$$J = J_i \prec J_{i-1} \dots \prec J_1 \prec Q$$

in the derivation graph. Note that  $J_k \cap J_{k-1} \neq \emptyset$  for all  $k \in \overline{1 \dots n}$ , and  $J_1 \cap Q \neq \emptyset$ . Let  $B_i$  be a binding query for  $J_i$ . Then, one can compute a binding pattern  $B_i$  for  $J_i$  as the conjunction between  $B_{i-1}$  (of  $J_{i-1}$ ) and a path (or union of paths) in  $J_i \cup J_{i-1}$ . A path has the origin in an input attribute of  $J_i$  and reaches an input attribute in  $J_{i-1}$  that occurs also in  $B_{i-1}$ .

**EXAMPLE 2.** Consider the query in Figure 6. The query asks for the titles that the Queen Victoria has had. The function  $h_1$  can

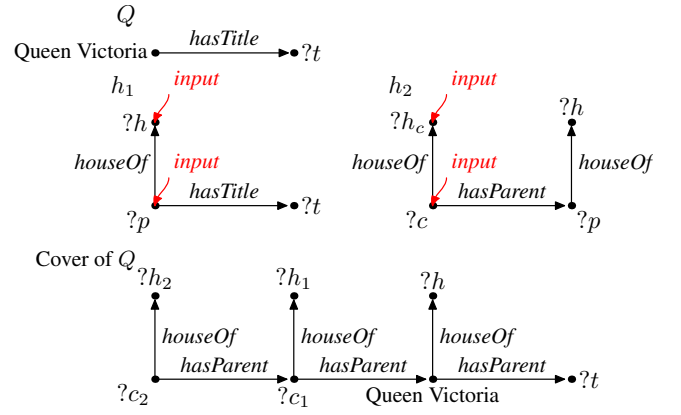


Figure 6: Lazy query construction.

be used to obtain the titles. Note that  $h_1$  takes as input the house (dynasty) and the name of the person. Let  $J_1$  be the partial instantiation for  $h_1$  so that  $\sigma_1(?p) = \text{"Queen Victoria"}$ . Assume that the house information is missing from the local base. However the function  $h_2$  can provide it. Let  $J_2$  be the partial instantiation for  $h_2$  so that  $\sigma_2(?p) = \text{"Queen Victoria"}$ . Assume that the local base contains some of the children names, but not their house attribute. Hence, the binding query  $B_2 = (?c_1, hasParent, \text{"Queen Victoria"})$  has answers. Therefore, the lazy construction continues to append another instantiation for  $h_2$ , as showed in Figure 6. Let  $J_3$  where  $\sigma_3(?p) = ?c_1$ , be the new instantiation. The recursive construction can continue until a descendant for whom the house attribute is stored in the local base is found (e.g. "Elisabeth II" of house Windsor), or no new generations are reached. Once a binding for all inputs is found, the evaluation proceeds bottom-up.

### 6.2 Quality of Web service calls

The operation that has the largest impact on the total execution time is the execution of Web calls. The algorithm should prioritize the execution of the calls in Web call composition that lead to a solution with a smaller cost. Consider an instantiation  $J_i$  so that  $J_i \prec \dots \prec J_1 \prec Q$ . Based on the *quality* of the functions  $f_i, \dots, f_1$  that correspond to the instantiations, we can estimate the quality of the instantiation  $J_i$ .

**DEFINITION 10 (QUALITY OF SERVICE).** The quality of service for a function  $f$  is a pair  $(p, t, a\vec{v}g)$ , where  $p$  is the probability for which the function  $f$  returns an not-empty answer,  $t$  is the average response time,  $a\vec{v}g$  is a vector that contains for each edge, the average number of corresponding triples in the results of the calls.

We define the *quality* is a partial instantiation  $J_i$  as a hierarchy of parameters:

$$QoJ(J_i) = \left( \prod_{k \in \overline{i \dots 1}} p_k, \sum_{k \in \overline{i \dots 1}} t_k, \prod_{k \in \overline{i \dots 1}} p_k \cdot avg_k \right)$$

Consider a Web call  $W_i$  of  $J_i$ . The first component of the  $QoJ(J_i)$  denotes the probability for which a complete list of calls  $W_i \prec \dots \prec W_1$  is computed if the bindings for the Web call  $W_{k-1}$  are chosen arbitrarily from the result of  $W_k$ . The second component is the time necessary to execute the Web calls  $W_i \prec \dots \prec W_1$ . The third component is the estimation for the total number of Web calls

corresponding to  $J_{i-1} \prec \dots \prec J_1$ , where  $avg_k$  is a scalar that denotes the average number of Web calls  $W_{k-1}$  that use as inputs, values that are output by  $W_k$ .

### 6.3 Algorithm

The F-RDF algorithm uses the lazy construction strategy described in Section 6.1 to construct the query cover. The algorithm checks for early bindings for the input attributes of the instantiations. For each partial instantiation, we keep the list of query bindings, and of their results. More precisely, for  $J$  we keep the list of tuples  $(B, Q_B, R_B, \mathcal{W}_B)$  where  $R_B$  denotes the list of answer of  $B$ , and  $\mathcal{W}_B$  represents the list of Web calls whose inputs are obtained in the result of  $B$ . The list of tuples is sorted according to the sub-query  $Q_B \subseteq B$ , from the most selective  $Q_B$  (with the largest number of edges) to the least selective one. The Algorithm 2 stores the partial instantiations in a priority queue denoted with  $L$ . The instantiations are partially sorted according to the next rule, and secondly, according to the quality of the instantiation  $QoJ$ .

**RULE 1 (PRECEDENCE OF WEB CALLS).** *If  $J_j \prec J_i$  and  $B_j$  and  $B_i$  are two binding queries corresponding to  $J_j$  respective  $J_i$ , so that  $Q_{B_j} \supseteq Q_{B_i}$ , then all the calls in  $\mathcal{W}_{B_i}$  are executed before the calls in  $\mathcal{W}_{B_j}$*

The rule makes sure that the information present in the local database is used to obtain fast the first results. Note that a Web call of  $J_j$  is pipelined with a Web call of  $J_i$  in order to be used in an answer of the query. As a consequence of the rule, we have the following property:

**PROPERTY 1.** *For a pipeline of calls*

$$W_1 \prec \dots \prec W_{i-1} \prec W_i \prec \dots \prec W_n$$

*so that their input values are the results of  $B_1, \dots, B_n$ , where  $Q_{B_1} \supseteq \dots \supseteq Q_{B_n}$ , if the input values of  $W_i$  are already in the knowledge base, then the calls  $W_k$  for  $k \geq i$  are executed first.*

The binding input of  $W_i$  can be present in the knowledge base by other means than the result of  $W_{i-1}$ . Usually, the same metadata is replicated on multiple Web sites. As observed in [11], many of the Web databases copy from one another. This property is important because a new result of the query is produced with only calling at most  $(n - i)$  Web calls in the pipeline.

---

**Algorithm 2** *F-RDF*(Funct  $F$ , Query  $Q$ , int  $MAX\_CALLS$ )

---

```

1: while  $L.notEmpty()$  &&  $calls \leq MAX\_CALLS$  do
2:    $J(B, Q_B, R_B, \mathcal{W}_B) = L.getFirst();$ 
3:   for  $(W \in \mathcal{W}_B)$  do
4:      $W.execute();$ 
5:   end for
6:   for  $(J_i(B_i, -, -, -)$  so that  $B_i \subseteq B \cup J)$  do
7:     update  $R_{B_i};$ 
8:   end for
9:    $LazyCoverExtension();$ 
10: end while
```

---

### 6.4 Properties

Our algorithm guarantees that it produces first the composition of function calls that take as input the constants in the query, and whose answers added to the knowledge base produce new answers for the query. Only after this sequence is produced, it considers the exhaustive lists of calls.

**PROPERTY 2.** *If there is a sequence of Web calls*

$$W_1 \prec W_2 \prec \dots \prec W_n$$

*so that the inputs of each  $W_i$  are in the result of a binding query  $B_i$  (possibly as the result of previous calls), then the sequence of calls is output by the algorithm before the exhaustive list of calls.*

*Proof Sketch:* For simplicity sake, assume that all functions have exact one input. We prove that a list of partial instantiations  $J_1 \prec J_2 \prec \dots \prec J_n$ , corresponding to the Web calls is added to the cover.  $J_1$  is added because  $B_1$  has no empty results. When the results of  $W_1$  is added to the knowledge base, then the query  $B_1 \cup J_1$  has answers. Then, there is a binding query  $B_2 = B_1 \cup J_2$  whose answers contains the input value for  $W_2$ . According to the principle used to construct the cover,  $J_2$  is added to the cover, and  $W_2$  will be then executed. The proof is then by induction.

In practice, such sequence of calls exists. Web sources define functions for both direct and inverse relationships between input and output. Hence, Web sites allow for navigation in the remote graph of resources following both senses of a relationship (edge).

In the following, we show that determining finite rewritings to a query is PSPACE-complete.

**THEOREM 1.** *Let  $Q$  be a query expressed by our query language (as defined in Section 3.2), and let  $F$  be the set of function definitions. Determining a finite query cover of  $Q$  with functions from  $F$  is PSPACE-complete in the size of  $G$ .*

*Proof Sketch:* For the lower bound, our proof is by reduction from the General Geography problem, which is known to be PSPACE-hard. As for the upper bound, we give an algorithm that computes a finite rewriting to the query in PSPACE. For a given GG graph with a designated node  $Q$ , we run a breadth-first search from  $Q$  and constructing for each edge  $(u, v)$  that we encounter at an odd level (assuming that the outgoing edges of  $Q$  are at level 1), a new function with output variable  $v$  and input variable  $u$ . Analogously, for each edge  $(v, z)$  that we encounter at an even level, we introduce a new function with input variable  $z$  and output variable  $v$ . The node  $Q$  represents our query. The intuition behind this reduction is that Player II can choose any possible output, and Player I has to respond with the appropriate input. It can be shown that there is a winning strategy for Player I if and only if there exists a cover for the query.

**THEOREM 2.** *The F-RDF produces the complete list of Web calls for an evaluation with an unbounded number of calls.*

*Proof Sketch:* The proof is based on the following observation: for any partial instantiation  $J_j$ , F-RDF will expand all instantiations  $J_i$  for which  $J_i \prec J_j$ . We call this property the exhaustive neighborhood search (ENS) property. The proof is by contradiction. Assume that there is a query cover  $J_1, \dots, J_k$  that cannot be found by F-RDF. Then there must be some partial instantiation  $J_i, 1 \leq i \leq k$  with  $J_i \prec J_j$  for some  $j, 1 \leq j \leq k$  that could not be discovered by the F-RDF. By the ENS property follows that  $J_j$  could not have been discovered by the F-RDF either. We can continue this argumentation recursively until we reach  $Q$ , which could not have been discovered either by the F-RDF. This is by the construction of F-RDF a contradiction.



	$Q_1$		$Q_2$		$Q_3$		$Q_4$		$Q_5$		$Q_6$		$Q_7$	
	Answ.	calls	Answ.	calls	Answ.	calls	Answ.	calls	Answ.	calls	Answ.	calls	Answ.	calls
DF	1	6	32	490	3	78	1	13	2	11	11	71	4	7
F-RDF (R)	1	0	28	409	5	268	1	11	26	107	23	154	4	3
F-RDF	1	0	42	80	5	238	1	5	26	34	23	48	4	2

Figure 8: Evaluation results

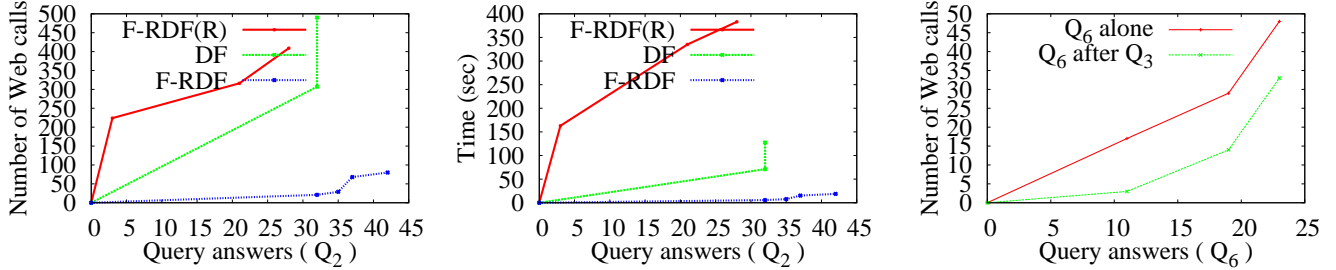


Figure 9: Distribution of answers during the evaluation

$Q_1$	"Frank Sinatra" bornOnDate ?birthday
$Q_2$	?author wrote ?book ?author hasWon "Nobel Prize in Literature" ?author isCitizenOf "Greece"
$Q_3$	"Frank Sinatra" hasChild ?child "Frank Sinatra" isMemberOf ?collaboration ?child isMemberOf ?collaboration
$Q_4$	"Reese Witherspoon" marriedTo ?spouse "Reese Witherspoon" actedIn ?movie ?spouse actedIn ?movie
$Q_5$	"Jane Austen" wrote ?book ?title titleOf ?book ?title titleOf ?movie ?actor actedIn ?movie
$Q_6$	"Frank Sinatra" hasChild ?child ?child sang ?song
$Q_7$	"Kristin Scott Thomas" actedIn ?movie ?title titleOf ?movie ?title titleOf ?book ?author wrote ?book

Figure 10: Queries

alone as input to any of the functions about the *sang* relationship. Instead, the function can accept only the *id* of a person, as defined by the MusicBrainz Web site. One can obtain the *id* using a prior call to another function; the function *getArtistId* provides the *id* of a singer when the singer name is given as input.

## 8.2 Results

**EXPERIMENT 1: ANSWERS & WEB CALLS** In Figure 8, we present the results of the seven queries showed in Figure 10. The evaluation in each case was bound to 1500 Web calls. In this setting, the number of answers returned by each algorithm serves as comparison metric. The F-RDF algorithm produces the largest number of answers in each case. The figure also shows the num-

Web Site	Function Name	(?x,?y)	avg.	$p$	$t$ (sec)
Music- Brainz	songsByArtistId	( $O,O$ )	15.22	0.99	0.93
	songBySongId	( $O,O$ )	0.89	0.99	0.57
	songByTitle	( $O,I$ )	5.06	0.99	0.92
last.fm	songBySongId	( $O,O$ )	1.00	0.08	0.46
	songsByTitle	( $O,I$ )	5.81	1.00	0.89

Figure 11: For the function edge ( $?x$ , *sang*,  $?y$ ), the average number of matching triples in Web service call results.

Web Site	Function Name	(?x,?y)	avg.	$p$	$t$ (sec)
isbndb	booksByAuthorId	( $O,O$ )	0.56	1.00	0.69
	booksByTitle	( $I,O$ )	1.46	0.99	1.41
AbeBooks	bookByTitle	( $O,I$ )	2.19	0.99	1.92
	booksByAuthor	( $I,O$ )	7.28	0.99	1.93
	booksByISBN	( $I,O$ )	6.10	1.00	1.07
Library- Thing	bookByTitle	( $O,I$ )	0.71	0.85	1.85
	bookByISBN	( $O,O$ )	0.31	0.96	0.86
	bookById	( $O,O$ )	1.85	0.74	0.80

Figure 12: For the function edge ( $?x$ , *wrote*,  $?y$ ), the average number of matching triples in Web service call results.

ber of calls after which all output answers are computed. We note that for the cases where the constants can be pushed as input parameters in Web calls e.g.  $Q_1$ , the number of Web calls leading to answers is small for all the algorithms. For queries where compositions of Web calls are necessary, the number of Web calls increases considerably, and the difference between F-RDF and DF becomes obvious. In Figures 9, the left most graph shows the relation between the number of answers and the number of Web calls that are executed in order to obtain the answers. The second graph show the relationship between the number of output answers and the evaluation time. The F-RDF algorithm converges fast to its total number of answers, and outputs the largest number of answers with respect to its competitors.

**EXPERIMENT 2** The second experiment illustrates the effect of

warehousing the data used in the evaluations of similar queries that preceded the current query. We show that the algorithm F-RDF makes uses of the local data in order to reduce the number of Web calls. Consider the following scenario. A user is exploring information about *Frank Sinatra*. Assume that he asks  $Q_3$  and then  $Q_6$ . We measure the number of calls necessary to execute  $Q_6$  after  $Q_3$  was executed, and we compare it with the case where only  $Q_6$  is executed. In Figures 9, the right most graph shows the results.

**EXPERIMENT 3** This experiment measures the precision and the recall of the query results. We consider 100 queries that ask for the books published by an author whose name is given. More than 98% of the output books were correct answers.

## 9. CONCLUSION

This paper has introduced a system for dynamically incorporating data from Web services into an RDF knowledge base, on demand for given user queries. We call this paradigm “Active Knowledge”, as it allows the knowledge base to actively and automatically complete or update its facts on entities or topics that the user is currently exploring. This happens transparently to the user, so that all browsing and querying of facts remains to be via RDF and SPARQL.

We emphasize again that our setting is different from a source-schema integration problem, as the Web services only provide encapsulated functions and do not expose any data schemas. The technical focus of this paper has been on efficiently generating sequences of function calls, with appropriately set parameters, to be executed by Web services based on judicious cost estimates. We could demonstrate, by experiments with a large knowledge based and prominent real-life Web services, that our algorithms obtain high recall with good answers delivered within the allowed cost budget for external service calls.

## 10. REFERENCES

- [1] S. Abiteboul, O. Benjelloun, and T. Milo. The Active XML project: an overview. *VLDB J.*, 2007.
- [2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
- [3] B. Amann, I. Fundulaki, M. Scholl, C. Beeri, and A.-M. Vercoustré. Mapping XML fragments to community Web ontologies. In *WebDB*, 2001.
- [4] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. DBpedia: A nucleus for a Web of Open Data. *The Semantic Web*, 2008.
- [5] M. Banko, M. J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni. Open Information Extraction from the Web. In *IJCAI*, 2007.
- [6] D. Berardi, D. Calvanese, G. D. Giacomo, R. Hull, and M. Mecella. Automatic composition of transition-based semantic web services with messaging. In *VLDB*, 2005.
- [7] C. Bizer, T. Heath, K. Idehen, and T. Berners-Lee. Linked data on the web (LDOW2008). In *WWW*, 2008.
- [8] A. Cali, G. Gottlob, and T. Lukasiewicz. A general datalog-based framework for tractable query answering over ontologies. In *PODS*, 2009.
- [9] K. C.-C. Chang, B. He, and Z. Zhang. Toward large scale integration: Building a metaquerier over databases on the web. In *CIDR*, 2005.
- [10] A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven web services. In *PODS*, 2004.
- [11] X. L. Dong, L. Berté-Equille, and D. Srivastava. Truth discovery and copying detection in a dynamic world. *PVLDB*, 2(1), 2009.
- [12] O. M. Duschka and A. Y. Levy. Recursive plans for information gathering. In *IJCAI (1)*, 1997.
- [13] R. Fagin, L. M. Haas, M. A. Hernández, R. J. Miller, L. Popa, and Y. Velegrakis. Clío: Schema mapping creation and data exchange. In *Conceptual Modeling: Foundations and Applications*, 2009.
- [14] D. Florescu, A. Y. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *SIGMOD Conference*, 1999.
- [15] D. Freitag and N. Kushmerick. Boosted wrapper induction. In *AAAI/IAAI*, 2000.
- [16] M. Friedman and D. S. Weld. Efficiently executing information-gathering plans. In *IJCAI (1)*, 1997.
- [17] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. D. Ullman, V. Vassalos, and J. Widom. The tsimmis approach to mediation: Data models and languages. *J. Intell. Inf. Syst.*, 8(2), 1997.
- [18] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4), 2001.
- [19] M. Jarrar and M. D. Dikaiakos. MashQL: a query-by-diagram topping SPARQL. In *ONISW*, 2008.
- [20] S. Kambhampati, E. Lambrecht, U. Nambiar, Z. Nie, and G. Senthil. Optimizing recursive information gathering plans in emerac. *J. Intell. Inf. Syst.*, 22(2), 2004.
- [21] G. Kasneci, F. M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum. NAGA: Searching and Ranking Knowledge. In *ICDE*, 2008.
- [22] I. Koffina, G. Serfiotis, V. Christophides, and V. Tannen. Mediating rdf/s queries to relational and xml sources. *Int. J. Semantic Web Inf. Syst.*, 2(4), 2006.
- [23] C. T. Kwok and D. S. Weld. Planning to gather information. In *AAAI/IAAI, Vol. 1*, 1996.
- [24] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB*, 1996.
- [25] B. Medjahed, A. Bouguettaya, and A. K. Elmagarmid. Composing web services on the semantic web. *VLDB J.*, 12(4), 2003.
- [26] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1), 2008.
- [27] A. Polleres. From SPARQL to rules (and back). In *WWW*, 2007.
- [28] R. Pottinger and A. Y. Levy. A scalable algorithm for answering queries using views. In *VLDB*, 2000.
- [29] K. Q. Pu, V. Hristidis, and N. Koudas. Syntactic rule based approach to Web service composition. In *ICDE*, 2006.
- [30] A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns. In *PODS*, 1995.
- [31] P. Senellart, A. Mittal, D. Muschick, R. Gilleron, and M. Tommasi. Automatic wrapper induction from hidden-Web sources with domain knowledge. In *WIDM*, 2008.
- [32] D. E. Simmen, M. Altinel, V. Markl, S. Padmanabhan, and A. Singh. Damia: data mashups for intranet applications. In *SIGMOD*, 2008.
- [33] F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: A Core of Semantic Knowledge. In *16th international World Wide Web conference (WWW 2007)*, New York, NY, USA, 2007. ACM Press.
- [34] K. P. Sycara, M. Paolucci, A. Ankolekar, and N. Srinivasan. Automated discovery, interaction and composition of semantic web services. *J. Web Sem.*, 1(1), 2003.
- [35] M. Technologies. The freebase project. <http://freebase.com>.
- [36] S. Thakkar, J. L. Ambite, and C. A. Knoblock. Composing, optimizing, and executing plans for bioinformatics web services. *VLDB J.*, 14(3), 2005.
- [37] V. Vassalos and Y. Papakonstantinou. Describing and using query capabilities of heterogeneous sources. In *VLDB*, 1997.
- [38] Word Wide Web Consortium. RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation 2004-02-10.
- [39] Word Wide Web Consortium. XSL Transformations (XSLT). W3C Recommendation 1999-11-16.
- [40] World Wide Web Consortium. SPARQL Query Language for RDF (W3C Recommendation 2008-01-15), 2008. <http://www.w3.org/TR/rdf-sparql-query/>.
- [41] F. Wu and D. S. Weld. Automatically refining the Wikipedia infobox ontology. In *Proc. of the Int. WWW Conf.*, 2008.
- [42] V. Zadorozhny, L. Raschid, M.-E. Vidal, T. Urhan, and L. Bright. Efficient evaluation of queries in a mediator for websources. In *SIGMOD Conference*, 2002.