Coupling Knowledge Bases and Web Services for Active Knowledge

Nicoleta Preda, Fabian M. Suchanek, Gjergji Kasneci
Thomas Neumann, Gerhard Weikum

March 30, 2009

MPI–I–2009–5–003 March 2009

Authors' Addresses

Nicoleta Preda
Max-Planck Institute for Informatics
Campus E1.4
66123 Saarbrücken
Germany

Fabian M. Suchanek
Max-Planck Institute for Informatics
Campus E1.4
66123 Saarbrücken
Germany

Gjergji Kasneci
Max-Planck Institute for Informatics
Campus E1.4
66123 Saarbrücken
Germany

Thomas Neumann
Max-Planck Institute for Informatics
Campus E1.4
66123 Saarbrücken
Germany

Gerhard Weikum
Max-Planck Institute for Informatics
Campus E1.4
66123 Saarbrücken
Germany

Abstract

We present ANGIE, a system that can answer user queries by combining knowledge from a local database with knowledge retrieved from Web services. If a user poses a query that cannot be answered by the local database alone, ANGIE calls the appropriate Web services to retrieve the missing information. In ANGIE, Web services act as dynamic components of the knowledge base that deliver knowledge on demand. To the user, this is fully transparent; the dynamically acquired knowledge is presented as if it were stored in the local knowledge base.

We have developed a RDF based model for declarative definition of functions embedded in the local knowledge base. The results of available Web services are cast into RDF subgraphs. Parameter bindings are automatically constructed by ANGIE, services are invoked, and the semi-structured information returned by the services are dynamically integrated into the knowledge base

We have developed a query rewriting algorithm that determines one or more function composition that need to be executed in order to evaluate a SPARQL-style user query. The key idea is that the local knowledge base can be used to guide the selection of values used as input parameters of function calls. This is in contrast to the conventional approaches in the literature which would exhaustively materialize all values that can be used as binding values for the input parameters.

Keywords

Web services, ontologies, intensional data, data integration

Contents

1 Introduction

Recent advances on automated information extraction [7] from textual and semistructured Web sources (e.g., Wikipedia) has enabled large-scale harvesting of entity-relationship-oriented facts to build large-scale knowledge bases. Projects like DBpedia [3], YAGO [23, 13], Freebase [24], KnowItAll [4], or Intelligence-in-Wikipedia [29] have successfully created semantic databases with many millions of facts about entities (e.g., persons, companies, movies, locations) and relationships (e.g., *bornOnDate*, *isCEOof*, *actedIn*, *shotAtLocation*). A convenient representation for these knowledge bases is the Semantic-Web data model RDF, and the data can be queried by SPARQL-like languages and interactively explored with user-friendly GUIs and visualization tools.

The knowledge stored in these databases is huge, but can never be complete and inevitably exhibits gaps that may irritate the user during interactive access. Consider for example a user who is interested in Frank Sinatra. Using the browser of the knowledge base, she has already found biographic information about Frank Sinatra, such as birthdate and birthplace. But when she is also looking for the albums of Frank Sinatra or for the movies that feature him, the information in the database is probably incomplete. Crawling additional Web sites on music and extracting the missing data is often infeasible because of site restrictions and because the site's information is continuously changing. Moreover, some knowledge needs are inherently ephemeral: for example, asking for the current rating of a movie (by averaging user reviews) or the chart rank of a song.

Fortunately, there is an increasing number of Web services that could fill the gaps in the database. There are Web services that deliver information about albums and music, books, movies and videos, etc. These services make their data available through an online API in a semi-structured format. The eConsultant lists a total of 122 public Web services[1] and Seekda even provides a search engine for Web services[2]. Note that it is practically infeasible to materialize all the data provided by these services. However, if the relevant data could be dynamically retrieved in the current user context, a much larger number of user queries could be answered. This is our vision of an *active knowledge base*. An active knowledge base is a dynamic federation of knowledge sources where some knowledge is maintained locally and other knowledge is dynamically mapped into the local knowledge base on demand.

An active knowledge base poses several challenges – even if the Web services are known to the system. First, different Web services may have to be combined in order to retrieve the desired results. In our example, a specific function of the MusicBrainz service has to be called first to obtain an identifier for Frank Sinatra, before another service function can be called to retrieve his albums. In full generality, multiple Web services from multiple Web sites may have to be combined with data that exists already in the database in order to construct the desired result. If there are multiple Web services that can deliver a certain piece of information,

[1]`http://webdeveloper.econsultant.com/web-services-api-services/`
[2]`http://seekda.com/`

these Web services may have to be prioritized by their response times or other service-quality properties. Finally, the system has to integrate the results from the Web services into the local database in a seamless manner. This poses a data cleaning challenge. All of these processes have to happen behind the scenes, so that the user still has the impression that all answers are returned by the local knowledge base.

Contribution This paper presents ANGIE, a system that can retrieve data from Web services on the fly whenever the local knowledge base does not suffice to answer a user query. In ANGIE, Web services act as dynamic components of the knowledge base that deliver knowledge transparently on demand.

We propose a model for declarative definition of functions embedded in the local knowledge base. The functions act as dynamic components of the knowledge base that deliver knowledge transparently on demand. Our framework helps integrating the semi-structured information returned by Web services dynamically into the knowledge base by combining different paradigms of data integration and techniques from data warehousing and mediation. Our key contributions are:

1. A clean model to represent Web services in a knowledge base, including their calling conditions.
2. An algorithm that determines all possible and valid combinations of Web services that could satisfy a query.
3. A system that dynamically retrieves requested information either from the knowledge base or from a composition of Web services, scheduling the calls according to their cost.

The current implementation uses YAGO [23] as local knowledge and has seamless connections to a variety of rich Web services on books (`isbndb.org`, `librarything.com`), movies (`api.internetvideoarchive.com`), and music (`musicbrainz.org`, `lastfm.com`). ANGIE comes with a light-weight desktop tool for querying, browsing, and visualizing the active knowledge base. Users can query data from local and dynamic sources in an arbitrary manner, without having to know where the data actually resides or is constructed.

1.1 Related Work

Recently, the ActiveXML [1] framework was proposed to use embedded function calls in XML data for data integration. In contrast to the present work, ActiveXML is centered around XML documents. In ActiveXML, no semantics is defined for the position of a call in the data, and the result is simply appended to the database.

Other work [22, 12] has centered on a mash-up of RSS and Web pages. In contrast to such approaches, ANGIE acts like a mediator system, where the query dynamically combines data from local and external sources on demand.

The approach of the Linking Open Data Project [5] is to connect data from Semantic Web sources in a global and distributed graph, using URIs and RDF specifications. Our framework instead aims at integrating dynamic components such as Web services, which have not been considered in the Linking Open Data Project. This task involves Web service composition [19], entity resolution on the fly [22] and schema mappings [2]. The authors of [21] have shown how Web forms can be wrapped into Web services. This work could actually be used to open up the world of the Deep Web to access by Web services – and thus, ultimately, to our approach.

There is ample work on data integration systems, but most prior work used an object oriented representation (e.g., [11]) or XML [30], which are both less adequate for the RDF knowledge bases that we consider.

Recently, the author of [17] showed how SPARQL queries can be translated to Datalog queries with negation. In the context of data integration, well known techniques *e.g.* [15, 18, 8], have been developed to address the problem of answering queries using views, where the views and the global schema are defined as Datalog rules. However, the two algorithms BUCKET [15] and MINICON [18] (an improvement over BUCKET), are not designed to work for views with limited access patterns. The first step in both algorithms is to identify views that are relevant to the query. This is not adapted to our need of both managing access-pattern limitations and allowing sequential chaining of services. These algorithms do not consider sources whose only role is to provide another service values for one of its bound parameters, even though this might be the only way to use this second service. Furthermore, recursive query plans are not considered. Another approach to answering queries using views for data integration purposes is the inverse rules algorithm [8]. It basically consists of rewriting a LAV (Local As View) system into a GAV (Global As View) system, by inverting each rule and introducing skolem terms. Recursive query plans are considered. Although this is an elegant solution for computing the results, it can not be aplied to our settings. The algorithm first computes the consequences of the rules (instantiatiates all possible function calls) and then uses a bottom-up algorithm to compute the answers. Other works (*e.g,* [20]), addressed query answering with views with binding patterns where all the sources are complete. They provide a single equivalent rewriting of the query. However, this solution is not suitable in our context of data integration where the sources can be incomplete. We are interested in modeling Web services because this seems to emerge as the standard for publishing information on the Web, in contrast to other works [25] that consider the case where the source itself has querying capabilities. In those settings the sources are described by infinite views.

2 Famework

2.1 Semantic Graph

In tune with recent work [23, 3, 13], we represent our knowledge base in the RDFS standard [26]. In RDFS, knowledge is modeled as a *semantic graph*. A semantic graph is a directed labeled graph, in which the nodes of are entities (such as individuals and literals) and the labeled edges represent relationships between the entities. We call such a graph a *semantic graph*. A fragment of a sample semantic graph is shown in Figure 2.1.

Formally, a semantic graph over a set of relations Rel and a set of entities $Ent \supseteq Rel$ 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 relations (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: *(artist, subclassOf, singer)*.

For our experiments, we used the semantic graph of YAGO[23]. YAGO is a knowledge base that has automatically been extracted from Wikipedia and WordNet[10]. It knows 2 million entities and contains 19 million facts about them. The facts include a class hierarchy of 200,000 classes. There are round 100 relationships. These include for example biographic relationships such as *bornIn* and *bornOnDate*, geographic information such as *hasCapital* and *locatedIn*, and information about movies such as *actedIn* and *hasProductionLanguage*. YAGO is also available in RDFS[1].

RDFS allows *reifing* a statement. That means that a statement itself can be an entity in the semantic graph. Suppose for example that we would like to reify the statement *(FrankSinatra, marriedTo, AvaGarner)*. We first choose an identifier for the statement (say, *s42*), which becomes an entity in the semantic graph. Then, each component of the statement is linked to the identifier as shown in Figure 2.2. This is possible, because relations are also entities in RDFS. This means that they can be nodes in the semantic graph. Note that even though *s42* is part of the semantic graph, the original statement does not have to be part of the graph.

2.2 Query language and semantics

We consider a subset of the SPARQL standard query language [28].

[1]at `http://mpii.de/yago`

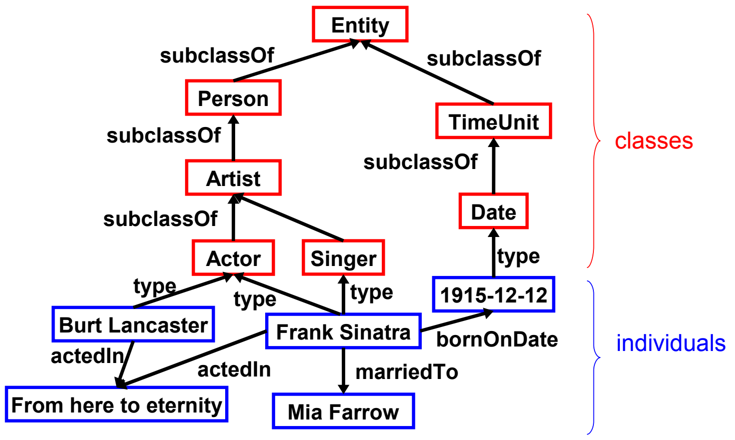

Figure 2.1: A semantic graph

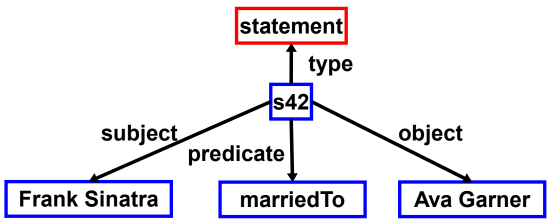

Figure 2.2: Reification

Definition 1 *A query over a set of variables Var for a semantic graph $G \subset Ent \times Rel \times Ent$ is a semantic graph $Q \subset (Ent \cup Var) \times (Rel \cup Var) \times (Ent \cup Var)$.*

Figure 2.3 shows two example queries. The first one asks for albums released by Frank Sinatra. The album is represented by a variable. The second query asks for Frank Sinatra's relationship to Ava Garner. In this case, the relation itself is a variable.

Definition 2 *An answer for a query q on a semantic graph G is a substitution σ for the variables of q, such that $\sigma(q) \subseteq G$.*

For instance, consider again the semantic graph shown in Figure 2.1. The query Q_2 has an answer in the semantic graph because there is the substitution $\sigma(\$r)$=marriedTo. $\sigma(Q_2)$=*(FrankSinatra, marriedTo, AvaGarner)* is a sub-graph of the semantic graph. There is, however, no answer to Q_1 in the current database. Therefore, a Web service has to be called.

2.3 Functions

In our ANGIE framework, the user can query for data that is not yet in the knowledge base. Rather, this knowledge is retrieved on the fly by calling Web services. Technically, we see a Web service as a function, which, given input parameters, returns output values. While the

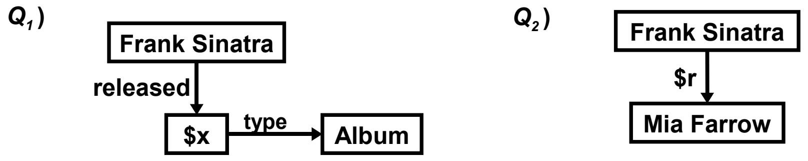

Figure 2.3: Two sample queries

local knowledge base is given extensionally, the knowledge provided by the functions is given intensionally. That is, the functions 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 3 *A function definition is a query $f \subset (Ent \cup Var) \times Rel \times (Ent \cup Var)$, where the edges are partitioned into pre-conditions and post-conditions.*

Intuitively speaking, a function definition specifies certain pre-conditions (i.e. edges) that have to be fulfilled before the function can be called. For example, one function may accept as input parameter the name of a pop star. Consequently, a pre-condition would be *($X,type,popStar)*. After the function has been called, new conditions are added. For example, the function may return an album of that pop star. Consequently, a post-condition would be *($X,hasAlbum,$Y)*, where *$Y* is instantiated by the function call. This way, the partition of edges implicitly determines a partition of the variables in V into input variables (those that appear in some pre-condition) and output variables (those that appear exclusively in post-conditions).

In practice, Web services define also optional input parameters. When the service is called, these parameters may be specified or omitted. In our representation, this can be modeled by providing multiple function definitions for the same Web service. Each function definition specifies a different set of preconditions. Graphical interfaces may allow defining functions with optional parameters and then compile them down to multiple function definitions.

Graph representation In our ANGIE 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, the variables names themselves become nodes in the semantic graph. Each reified edge is connected to the function identifier by an edge with *preconditionOf* or *postconditionOf*. Figure 2.4 a) shows an example. The function *getAlbums* has the pre-condition that *$X* be a singer. It has as post-conditions that *$X* released the album *$Y*. To simplify our illustrations, we use blue color for reified pre-conditions and red color for reified post-conditions, as shown in Figure 2.4 b).

This way, the function definitions are integrated completely into the semantic graph. Thereby, function definitions become first class citizens of the knowledge base. Thus, it is possible to query the knowledge base for functions that have certain properties.

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 the ANGIE model. The role of extensionally defined predicates in Datalog is taken over by concrete

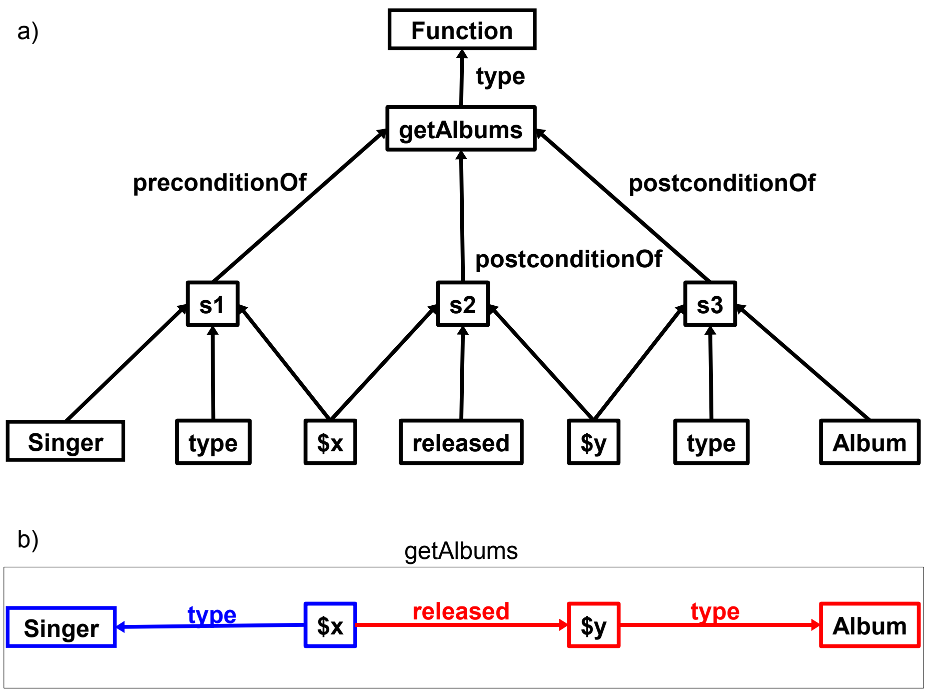

Figure 2.4: A function definition

edges in the semantic graph in ANGIE. The role of intensionally defined predicates is taken over by *intenstional rules*. An intensional rule is given by an ordinary function definition, but it lacks an underlying Web service. Furthermore, all variables that occur in post-conditions must occur in pre-conditions. If such a function-composition rule gets called, the post-conditions are added to the knowledge graph without reference to any Web service. Since the pre-conditions act as conjunctive conditions, intensional rules have the same expressive power as domain-restricted Horn clauses with binary predicates in Datalog.

For instance, consider the semantic graph G in Figure 2.5. It is known that Frank Sinatra is a singer and it is known that *singer* is a subclass of *person*. Intuitively, this means that Frank Sinatra is also a person. However, this implicit information has not yet been materialized in the database. To do so, it suffices to specify the intensional rule f. f has as pre-conditions that *$individual* is an instance of some class *$sub* and that *$sub* is a subclass of some other class *$super*. Its postcondition is that *$individual* is an instance of *$super*. (Note that this is a typical Horn clause.) When f gets "called" (i.e. evaluated), the variables in the pre-conditions get instantiated and the post-condition is added to the database.

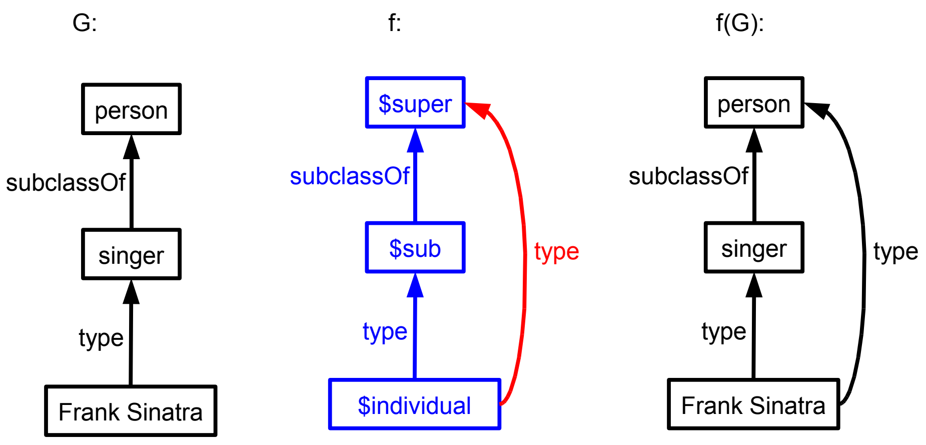

Figure 2.5: An intensional rule

3 Query Rewriting Problem

The ANGIE knowledge base consists of an extensional part, stored locally, and an intensional part, accessible by function calls. Given an user query, the problem is to evaluate the query using the local knowledgebase and the set of functions defined in ANGIE.

Note that the functions have limited access patterns: values for the input parameters must be provided in order to obtain bindings for the output parameters. This makes the problem difficult because, in order to obtain the desired values for the output, one should provied the meaningful input values. For instance, the function call f_1 can be used to retrieve the albums created by Frank Sinatra only if the *id* of Frank Sinatra is provided as input. One solution is to enumerate all the values that can be provided as input to a function, and to compute all the consequences of a function. This solution does not work in practice due to the large volumes of data generated by all the calls, which can not be transferred using the limited bandwith connections in the Internet.

Our approach is to treat the functions as intensional components of the ontology and represent them within the ontology as described in Section 2.3. We see the functions as active components of the ontology: a call to a function provides, on demand and when executed, bindings for the output parameters. Furthermore, the output of the already executed calls enrich the ontology and can be directly read from the local ontology later, for evaluating other user queries. For a given user query expressed in a subset of the SPARQL query language (see Section 2.2) one must compute the set of functions whose results might be used in order to answer the query. We may see the initial query plus the set of function calls as a new query defined in an extended SPARQL language that allows for declarations of function calls. This is similar to the classical approach in data integration frameworks where the initial user query is translated into a new query denoted the *rewriting query*. Specific to our rewriting problem, we encounter the following difficulties:

Access Limitations The great challenge in this context is to determine the set of functions that must to be called in order to answer a user query and more difficult, to determine the actual values that must be provided in the input of the calls.

Function composition Note that the process of computing the rewriting may be dynamic, as the result of one call may produce data that can be used as input for other calls.

Incompleteness of Web sources We may see ANGIE as a data integration system where the global mediated schema consists of the classes and the relations defined in it. The functions (defined as parameterized queries over the global mediated schema) can be seen as views (named queries) over external data sources (Web services). In this respect, the functions are similar to the views with limited access patterns described in [20, 8]. The data sources on the Web are usually incomplete, as they do not cover the entire domain of a class. In such settings, one can compute only *maximal contained query rewritings*. A query Q_1 is said to be contained in a query Q_2, denoted by $Q_1 \sqsubseteq Q_2$, if for all databases the D, the set of tuples for Q_1 is a subset of those computed for Q_2 i.e. $Q_1(D) \subseteq Q_2(D)$. A contained query rewriting provides a part, but not all the answers of a query. A *maximal contained rewriting* for a query, with respect to a language, is a query so that no other rewriting contains it. As shown in [9], there are several important cases in which a maximal contained rewriting of a query can only be found if recursive Datalog rewritings are considered. In [14], the authors show that for views with limited access patterns there might be *no* bound on the size of the rewriting. However, as shown in [9], a finite rewriting of the query, albeit a recursive one can be computed.

Our rewriting problem As opposed to other query rewriting algorithms that aim at computing the complete set of answers, our goal is to produce the first answers quickly. We are interested to determine first which calls are more likely to return useful data for the query. The solution must scale to a large number of functions defined in our system.

Usually, expensive query rewritings must be executed in order to guarantee that the complete set of answers is computed. For practical reasons, in order to limit the resource consumption (bandwidth consumption caused by a large number of remote Web calls), we limit the processing of the query using a timeout for the query processing. For an infinite value of the timeout, the algorithm must produce the complete answers with respect to the data sources and to the query language used to define the functions (views) and the query.

4 General Answer Model

This section explains how the query engine can answer a query from the user. For this purpose, we first explain how the query can be translated into function calls. This translation is called a *rewriting*. Then, we explain how the function calls can be scheduled and executed.

4.1 Rewritings

Formally, a family of calls to a function with certain parameters is defined as a *function instantiation*:

Definition 4 *A* function instantiation *for a function definition $f \subset (Ent \cup Var) \times Rel \times (Ent \cup Var)$ is a complete substitution $\sigma : Var \rightarrow Ent \cup Var'$, where Var' is a fresh set of variables.*

The function instantiation will instantiate some variables of the function definition with entities. Other variables of the function definition are simply given new names. It is necessary to give them new names, because the same function definition may be instantiated multiple times and naming conflicts have to be avoided.

We will now explain how a query can be rewritten to a knowledge base query and a set of function instantiations. For uniformity, let us also consider the triples in the database as given by a function. For this purpose, we introduce an artificial function f_{db}. f_{db} is the function definition that consists of no pre-conditions and one single post-condition edge, in which all three components are variables. Whenever this function is ”called” it issues an SPARQL query to the knowledge base.

Definition 5 *A* rewriting *for a query Q with a set of function definitions F is a set of function instantiations $\sigma_{f_1}, ...\sigma_{f_n}$ for the functions $f_1, ..., f_n \in F \cup \{f_{db}\}$ on a common set Var' of fresh variables, so that*

1. *for each precondition edge pre of any function f_i,*

$$\exists f_j, post \text{ postcondition of } f_j : \sigma_{f_j}(post) = \sigma_{f_i}(pre)$$

2. $\forall q \in Q \exists f_i, post$ *postcondition of* $f_i : \sigma_{f_i}(post) = q$

A rewriting is a set of function instantiations such that each edge in the query and each pre-condition of a function instantiation is satisfied as post-conditions of another function instantiation. The rewriting specifies which functions have to be called with which parameters in order to find bindings for the variables in the query. A call to f_{db} corresponds to a query to the local knowledge base. The simplest rewriting uses only function instantiations of f_{db}. In this case, the rewriting is identical to the query (with the variables renamed).

Algorithm 1 DF(F, Q, R)

```
Input: $F$: the set of function definitions
Input: $Q$: a list of query edges
Input: $R$: the current function composition
 1: if $Q = \emptyset$ then
 2:     Output $R$
 3:     return
 4: end if
 5: $q := Q.poll()$
 6: for all $f \in \{f_{db}\} \cup F$ do
 7:     for all post-conditions $p \in f$ do
 8:         if $\exists\ \sigma :\ \sigma(p) = q$ then
 9:             $\sigma' :=$ a substitution for the free variables in $f$
10:             $\sigma'' := \sigma'\ o\ \sigma$
11:             DF($F, Q \cup \sigma''(pre(f)) \setminus \{q\}, R \cup \{(f, \sigma'')\}$)
12:         end if
13:     end for
14: end for
```

Algorithm 1 computes and outputs the rewritings for a query in the spirit of the chronological backtracking strategy used in Prolog. The algorithm takes as input a set F of function definitions, a list Q of query edges and the current rewriting R. In the initial call, $R = \emptyset$. The algorithm implements a recursive deep-first search on all possible rewritings. In each recursion step, one edge is removed from Q and one function instantiation is added to R.

Let us look at the algorithm in more detail. Suppose for example that the query Q asks for the albums released by Frank Sinatra. Suppose that we have 3 function definitions: $f1$ maps an artist to its id in the MusicBrainz world. $f2$ maps the id to the albums released by the artist. $f3$ is the function-composition rule from Figure 2.5. Figure 4.1 depicts this situation.

The algorithm first checks whether there are any more query edges left. If no more edges are left, the rewriting is output and the algorithm returns. In our case, there is one query edge left. The algorithm picks this query edge (q). The algorithm considers all possible function definitions. For each function definition f, it checks all post-conditions p. It tries to find a substitution that satisfies q, i.e. a substitution σ such that $\sigma(p) = q$). In our case, $f3$ can satisfy q with σ(*$artist*)=*FrankSinatra* and σ(*$album*)=*$x*. The algorithm expands σ to a proper function instantiation σ'' that renames also the remaining free variables in f. In our case, σ''(*$id*)=*$id/0*. Then, the algorithm calls itself recursively. For the next recursion step, q is removed from Q, because q is satisfied. However, the preconditions of f become new members of Q. Figure 4.2 shows the removed edges in blue and the added edges in red. In the next steps, the algorithm tries to satisfy the edges marked with *hasMBID* and *idReleased*. In the figure, the algorithm chooses f_{db} to satisfy the edge marked with *hasMBID*. This does not add any pre-conditions to the query. Note that the algorithm could have chosen f_1 as well for this edge. We will discuss the selection policy in the next section. In the next step, the algorithm chooses f_2 to satisfy the edge marked with *idReleased*. This adds one pre-condition, which can be satisfied by f_{db}. The result is a rewriting that “covers” the initial query completely.

The algorithm implements a shortcut for function definitions that have more than one post-condition: If a post-condition of a function definition matches a query edge and if the function

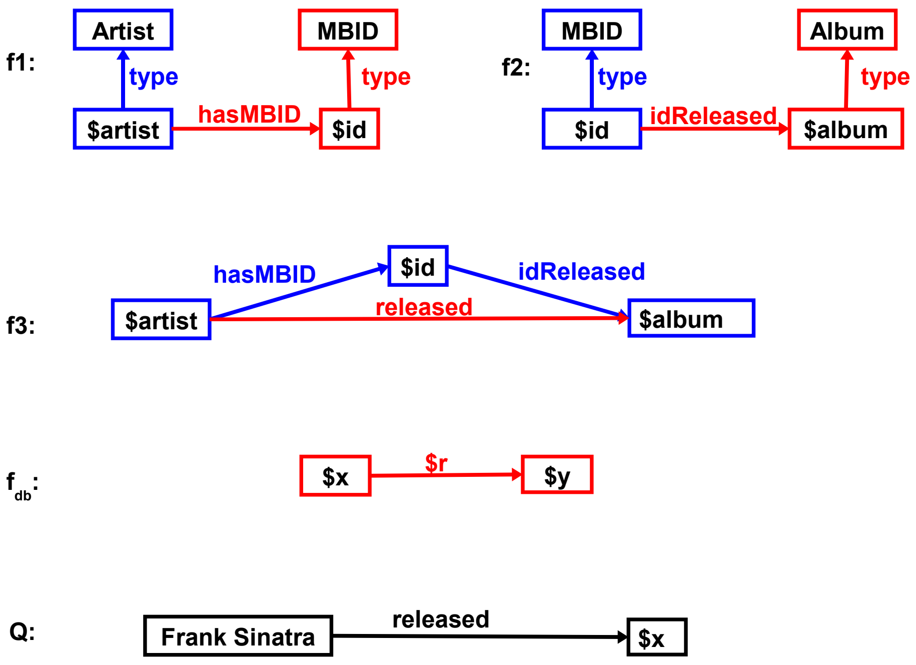

Figure 4.1: Function definitions and query

definition already appears in the rewriting, then no new function instantiation is needed.

4.2 Cost model

The operation that has the largest impact on the total execution time is the execution of calls to Web services.

Hence, the cost of executing a user query depends of the *quality of σ instantiations*. The algorithm should prioritize the execution of σ instantiations so that the instantiations that lead to a solution with a smaller cost are executed first. We consider the response time as the cost of executing a query.

The quality of a σ_f instantiation depends in turn of (i) the Quality Of Service f, and of (ii) the selectivity of the values used as binding values for the input parameters.

Definition 6 *The quality of a service f is a vector (p, t) where the parameter p represents the probability with which f returns a non empty answer for an arbitrary call that satisfies the pre-conditions, and t is the response time divided by the number of post-conditions returned in response.*

p is a measure of the incompleteness of the source implementing f. For the database function f_{db} we assign the the quality of service $(1, 0)$. This corresponds to the optimistic approach which considers that the local knowledge is complete. Furthermore, indeed the response time can be approximated with 0, as the response time for a the database access is orders magnitude smaller than that of a service call. For the other functions in the systems, the two values are learned from the the history log of the calls.

The other factor that characterizes the quality of a function instantiation is the selectivity of input values.

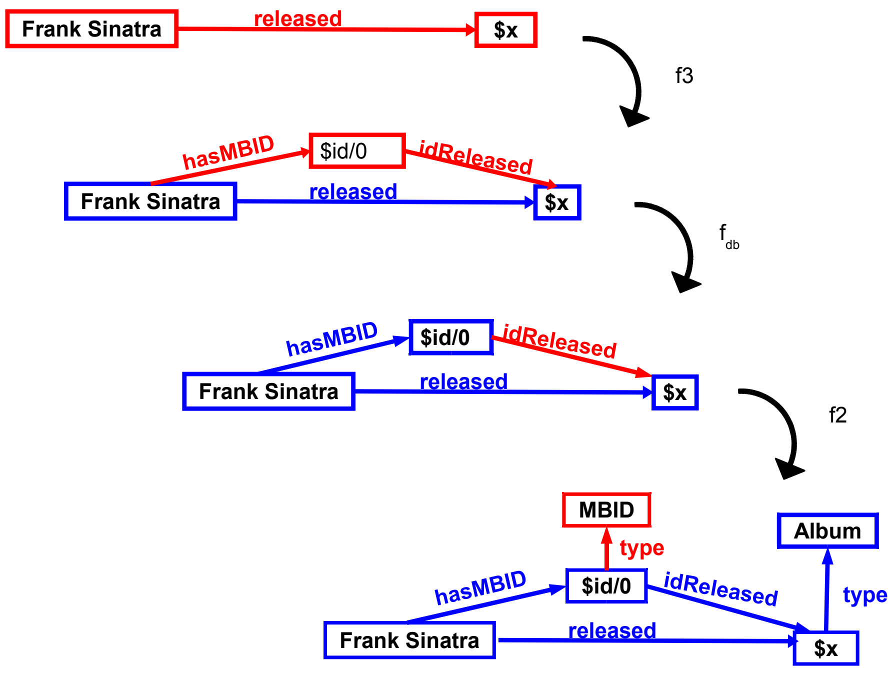

Figure 4.2: Rewriting the Query

We introduce a *cost function* c, which maps a function instantiation σ_f to a real value that reflects the expected cost of using the σ_f in the evaluation. If an instantiation σ_f uses as pre-conditions the post-conditions of another function instantiation $\sigma_{f'}$, then one must have an estimation of the number of post-conditions produced by the evaluation of $\sigma_{f'}$ in order to determine the number of calls corresponding to σ_f. Similarly, if f' is a database function, then the values returned as postconditions of $\sigma_{f'}$ can be computed directly from the database. If f' is a Web function, then one can only estimate the number of results. Such statistics can be computed and maintained for each function, based on the history of the calls.

Because the sources in the Internet are incomplete, not all the calls yield answers for valid input parameters. A call that produces no result just consumes resources. As a measure of the incompleteness of a source, every function has associated the parameter p_f representing the probability with which a call to that function returns an answer.

Definition 7 *The cardinality of the result generated by all the calls corresponding to σ_f is given by the expression:*

$$|Output(\sigma_f \mid_{post})| = p_f * \mid Input(\sigma_f) \mid * \mid Output_f \mid_{post}|$$

where $|Output(\sigma_f \mid_{post})|$ denotes the cardinality of postconditions post *produced by the instantiations of σ_f, $\mid Output_f \mid_{post}|$ is an estimation of the average postconditions* post *produced by calls to f, $\mid Input(\sigma_f) \mid$ denotes the number of calls corresponding to σ_f.*

Definition 8 *The total cost of executing the instantiation σ_f is:*

$$c(\sigma_f) = \mid Input(\sigma_f) \mid * t + (\textstyle\sum_{post} |Output(\sigma_f \mid_{post})|) * t_u$$

where $\mid Input_{\sigma_f} \mid$ denotes the number of binding values for the input parameters defined in σ_f and t stands for the time to execute one call and t_u is cost per transferred unit of data.

t represents the time necessary for establishing the connection with the remote Web service (which is inherent to each call), while t_u represents the cost necessary to transfer a unit of data (one post-condition).

A issue here is that function results may have have a power-law nature, i.e. lots of result sets have a single entry, many two entries and there are very few result set that are very large. Such power law distributed cardinality will influence the quality of the expected value for predictions. The quality of the prediction increases, if for equivalence classes i.e., they have similar properties.

Domain aware function definitions There are some cases where one knows precisely the set of values that can be used as input parameters for some Web function. For instance, consider a Web site that publishes all the authors for which it contains informations, and makes them available in the Internet. Consider also that the Web site provides a Web service that returns the books published by a given author. Note that for all the authors in the published list of authors, the probability to obtain a valid answer is $p_f = 1$. In order to discriminate between the case in which the function is executed for an author in the retrieved set of authors and the case in which the function is executed for an arbitrary entity, we can procede as follows. We introduce a new class grouping all the authors in the retrieved list of exact authors, and we specify in the definition of the function (from the ontology) that the input should have as type the newly introduced class, and we let $p_f = 1$.

4.3 Analysis of the algorithms

In this section we study the query rewriting solution in ANGIE. In [14], 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 two function f_1, and f_2 and the query Q defined in an equivalent datalog program:

$$\begin{array}{rcl}
f_1^{bf}\text{ (X,Y)} & \text{:-} & \text{Similar(X,Y), Artist(X), Artist(Y)} \\
f_2^{bf}\text{ (X,Y)} & \text{:-} & \text{Artist(X), Album(Y), created(X,Y)} \\
\text{Q(X)} & \text{:-} & \text{Artist(X), Created(X,'My Way')}
\end{array}$$

The function f_2 returns the albums released by an artist provided as input. Hence, one can use the function f_2 for all the artists in the ANGIE database, and filter out those who created no album labeled *'My Way'*. Furthermore, many more artists could be discovered if the function f_1 is called subsequently for artists in the ANGIE database and recursively for artists returned in the answer of f_2. Note that the chain of recursive calls to f_2, necessary to discover new artists, depends of the configuration of the data and is not bound by the size of the query. Furthermore, some of Web services in the Internet implement functions that generate domains consisting of infinite number of values. For instance, consider a function that given a year, returns the next year when a total eclipse of the sun is produced.

A issue is the number of function instantiations that should be considered for computing the complete set of answers. The following theorems shows that the complexity is exponential.

Theorem 1 *Determining all sets function instantiations that satisfy the pre-conditions of a function f, is EXPTIME-hard.*

As a direct consequence, we can conclude that determining all function instantiation that must be generated in order to obtain all query answers, is EXPTIME-hard. One can see the query as a function where all the edges are marked as pre-conditions.
Proof: The proof is by reduction from 2-Player Corridor Tiling game [6]. The setting of the game is given by a tuple $(T, C_v, C_h, \hat{t}, \hat{b}, n)$. T is a set of domino tiles. $C_v, C_h \subseteq T \times T$ represent vertical and horizontal constraints respectively, n is an even natural number, and $\hat{t} = (\hat{t_1}, ..., \hat{t_n})$ and $\hat{b} = (\hat{b_1}, ..., \hat{b_n})$ are n-tuples of tiles.

A Corridor Tiling is a mapping $\lambda : \{1, ..., m\} \times \{1, ..., n\} \rightarrow T$, where m is again a natural number. Furthermore, it must hold that $(\lambda(1, 1), ..., \lambda(1, n)) = \hat{t}$, and $(\lambda(m, 1), ..., \lambda(m, n)) = \hat{b}$. A Corridor Tiling is correct iff for every $i = 1, ..., m$ and every $j = 1, ..., n-1$, $(\lambda(i, j), \lambda(i, j+1)) \in H$ and for every $i = 1, ..., m-1$ and every $j = 1, ..., n$, $(\lambda(i, j), \lambda(i+1, j)) \in V$.

The 2-Player Corridor Tiling game is played by two players who place tiles in turn on a $\mathbb{N} \times \mathbb{N}$ board. The second player starts the game. While the first player tries to construct a correct Corridor Tiling, the second player tries to prevent it, by working against the strategy of the first player. The first player wins, when he manages to construct a correct Corridor tiling, or when the second player has no other possibility than placing a tile which violates the rules in C_v or C_h.

It is known that the problem whether there exists a winning strategy for the first player is EXPTIME-hard. We call this problem the 2PCT problem.

Now consider the tree representing all possible instantiations of a function f. At the root of this tree there is a dummy node. The children of the dummy node are all postconditions of the function f. Each postcondition node has as children all preconditions that need to be fulfilled, in order to fulfill the postcondition. Each precondition node has as children all postcondition of the functions that can fulfill the precondition, and so on. If there exists an instantiation of the function f that binds all its postconditions, then every path in the tree has to represent a chain of function calls that binds the second node of that path when starting from the root (note that this node is one of the postconditions of f). For now, we call this tree the instantiation tree.

We show in the following that any 2PCT problem can be reduced to our problem of determining whether there exists a binding instantiation for f in polynomial time.

Consider a 2PCT problem represented by $(T, C_v, C_h, \hat{t}, \hat{b}, n)$.

For each $\hat{t_i}, \hat{t_{i+1}}$ and each $\hat{b_i}, \hat{b_{i+1}}$, for an odd $i < n$, we introduce the postconditions $post_{\hat{t_i}}$ and $post_{\hat{b_i}}$ as well as the preconditions $pre_{\hat{t_{i+1}}}$ and $pre_{\hat{b_{i+1}}}$, such that $pre_{\hat{t_{i+1}}}$ has to be fulfilled in order to fulfill $post_{\hat{t_i}}$ and $pre_{\hat{b_{i+1}}}$ has to be fulfilled in order to fulfill $post_{\hat{b_i}}$. For all constraints of the form $(c, c_{j_i}) \in C_h$ and $(c', c_{j_i}) \in C_v, 1 \leq j_i \leq k \in \mathbb{N}$, we introduce the postcondition $post_c$, and the preconditions $pre_{c_{j_i}}$ such that there is only one $pre_{c_{j_i}}$ that can lead to the fulfillment of $post_c$. Furthermore, we introduce the precondition pre_c, and the postconditions $post_{c_{j_i}}$ such that any $post_{c_{j_i}}$ that can lead to the fulfillment of pre_c. For every constraint $(c, c_{j_i}) \in C_v$ such that there is no constraint of the form $(c', c_{j_i}) \in C_h$, we introduce the dummy precondition $\#_c$ and the post conditions $post_{c_{j_i}}$, such that all $post_{c_{j_i}}$ can fulfill $\#_c$. Finally, we turn the precondition $pre_{\hat{b_n}}$ into an f_{db}, a simple function call that can be answered directly from the local database. Our function f is given by all the postconditions that fulfill the dummy precondition $\#_{\hat{t_1}}$.

It is obvious that the above construction of preconditions and postconditions can only lead to an instantiation tree that indirectly represents the strategy of the first player in the 2PCT game. Hence, there exists a winning strategy if and only if there is a binding instantiation that fulfills all postconditions of f.

5 System architecture

The overall architecture of the ANGIE system is illustrated in Figure 5.1. The system uses the existing YAGO ontology [23], which consists of 2 million entities and 20 million facts extracted from various encyclopedic Web sources. In addition, ANGIE includes a built-in collection of function definitions for the following Web services: MusicBrainz, LastFM, Library Thing, ISBNdb, and IVA (Internet Video Archive). In our envisioned long-term usage, the function definitions would either be automatically acquired from a Web-service broker/repository or they could be semi-automatically generated by a tool, e.g., [2].

Query Translation Module. This is the core component of the ANGIE project. The module takes as input a user query, and translates it into a sequence of function compositions. Each function composition takes the form of an extended SPARQL query. In such a query, calls to the database function f_{db} are simple triples. Calls to other functions are embedded function-call declarations. The following sample query retrieves the id of Frank Sinatra from the knowledge base and his albums from Music Brainz:

```
PREFIX y :  <http://mpii.de/yago/resource/>
SELECT ?album WHERE {
     y:Frank_Sinatra y:hasMBId ?id
     FUNCTION(getMBAlbums, ?id, ?album)
}
```

RDF-3X processor. The generated SPARQL queries are sent to the RDF-3X processor [16]. The processor has been modified to accommodate the Web service calls. It is also responsible for scheduling the execution of the function calls. The calls are executed via the *Mapping Tool* (discussed below), which is in charge of remote invocation of the Web services. The Mapping Tool responds to the processor with the list of triples representing the answers of the calls. The RDF-3X processor combines the triples from the local knowledge base and the triples received from the mapping tool to produce a uniform output. The query translation and the query execution are interleaved. The translation module continuously sends SPARQL queries to the RDF-3X processor, which responds with new results.

The Mapping Tool. This component executes the Web service calls. It mediates between the function declarations in the knowledge base and the schema of the XML documents that the function call returns. For this purpose, every function has two mappings associated with it: The *lowering mapping* defines how the pre-conditions of a function are translated to the parameters of a REST (or SOAP) call. The call is sent to the remote site that provides the Web service. ANGIE supports the parallel execution of multiple calls. A call will yield values for the output variables in an XML fragment. The *lifting mapping* defines how the XML nodes in the answer are mapped to entities in the knowledge base. We use the XSLT standard [27] for this purpose. The entities can then be handled by the RDF-3X processor.

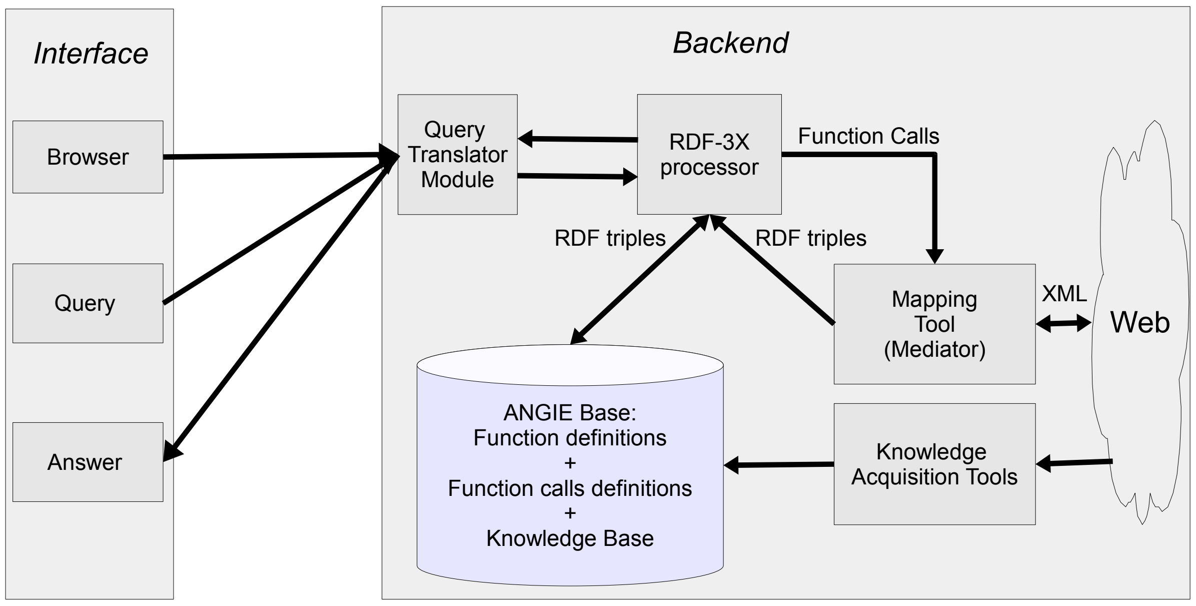

Figure 5.1: System architecture of ANGIE

User Interface. The user interface allows the user to query the knowledge base in the language described in Section 2.2. Queries are sent to the query translator module and answers are retrieved from there. Furthermore, the user can also display the knowledge base as a hyperbolic graph. One exploration step in this GUI retrieves and visualizes the neighborhood around a given entity. Such a browsing step translates into a simple query that retrieves the neighbors of that entity.

6 Conclusions

This work presents the advancement in the project ANGIE. The ANGIE project aims at building a system that can retrieve data from Web services on the fly whenever the local knowledge base does not suffice to answer a user query. In ANGIE, Web services act as dynamic components of the knowledge base that deliver knowledge on demand. To the user, this is fully transparent; the dynamically acquired knowledge is presented as if it were stored in the local knowledge base. We have developed a model for declarative definition of functions embedded in the local knowledge base. The (REST or WSDL) interfaces of available Web services are cast into such functions. Parameter bindings are automatically constructed by ANGIE, services are invoked, and the semi-structured information returned by the services are dynamically integrated into the knowledge base. Our approach is to treat the functions as intensional components of the knowledge base and represent them within the knowledge base itself. Executing a function provides bindings for its output parameters, filled in on demand by the invoked Web services. These output are added to the knowledge base and can be directly read later, for evaluating other user queries. A function definition is a parametrized query, represented in the form of a semantic graph (edges are relationship names and the nodes are variables or entities). Furthermore, the edges are partitioned into input pre-conditions and output post-conditions. The pre-conditions have to be fulfilled before the function can be called (by appropriate parameters filled in by ANGIE before external services are invoked). The post-conditions correspond to output parameters of the invoked services. This representation naturally extends the model of the Semantic knowledge.

Given a user query that cannot be directly or not completely answered by the local knowledge base alone, the challenge is to determine the set of functions that must to be called in order to provide answers. This in turn entails determining the actual values that must be provided as inputs. Moreover, a big difficulty that often arises is the need for composing functions. This may result from the specifics of a services interface or from the limitations regarding result sizes or other forms of incompleteness. For example, one service method may take a book title as input parameter and return an ISBN as its output, and there may be a second method for providing the author, publisher, etc. based on the ISBN as input, and a third method may have to be used to return the sales rank for a given pair of book and publisher names. Thus, many information requests can be satisfied only by composing several of these functions, and ANGIE needs to construct the appropriate parameter bindings. This rewriting a knowledge-base query into a function composition is dynamically performed step by step, as the result of one call may produce data used as input for subsequent calls. In contrast to more traditional query rewriting algorithms that aim at computing a complete set of answers, our goal is to produce the first answers quickly. Therefore, we need to estimate which calls are more likely to return useful data for the query, and at which execution costs.

We have developed a query rewriting algorithm that determines one or more function composition that need to be executed in order to evaluate a SPARQL-style user query. The key idea is that the local knowledge base can be used to guide the selection of values used as input parameters of function calls. This is in contrast to the conventional approaches in the literature which would exhaustively materialize all values that can be used as binding values for the input parameters.

A prototype system has been built, and we are conducting an experimental evaluation. Our system consists of several components: (i) a rewriting module implementing the algorithm outlined above, (ii) the existing RDF-3X engine [16] that carries out the execution of the rewritings and uniformly integrates the data from the local knowledge base and the data returned by the Web service calls, (iii)a mapping tool that maps the XML schemas of the XML data returned as service results onto the RDF model of the knowledge base.

Bibliography

[1] S. Abiteboul, O. Benjelloun, and T. Milo. The Active XML project: an overview. *VLDB J.*, 2007.

[2] B. Amann, I. Fundulaki, M. Scholl, C. Beeri, and A.-M. Vercoustre. Mapping XML fragments to community Web ontologies. In *WebDB*, 2001.

[3] 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.

[4] M. Banko, M. J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni. Open Information Extraction from the Web. In M. M. Veloso and M. M. Veloso, editors, *IJCAI*, pages 2670–2676, San Francisco, USA, 2007. Morgan Kaufmann.

[5] C. Bizer, T. Heath, K. Idehen, and T. Berners-Lee. Linked data on the web (LDOW2008). In *WWW*, 2008.

[6] B. S. Chlebus. Domino-tiling games. *J. Comput. Syst. Sci.*, 32(3):374–392, 1986.

[7] A. Doan, L. Gravano, R. Ramakrishnan, and S. V. (Eds.). Special issue on managing information extraction. *ACM SIGMOD Record 37(4)*, 2008.

[8] O. M. Duschka, M. R. Genesereth, and A. Y. Levy. Recursive query plans for data integration. *J. Log. Program.*, 43(1):49–73, 2000.

[9] O. M. Duschka and A. Y. Levy. Recursive plans for information gathering. In *IJCAI (1)*, pages 778–784, 1997.

[10] C. Fellbaum, editor. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[11] 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):117–132, 1997.

[12] M. Jarrar and M. D. Dikaiakos. MashQL: a query-by-diagram topping SPARQL. In *ONISW*, 2008.

[13] G. Kasneci, F. M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum. NAGA: Searching and Ranking Knowledge. In *ICDE*, 2008.

[14] C. T. Kwok and D. S. Weld. Planning to gather information. In *AAAI/IAAI, Vol. 1*, pages 32–39, 1996.

[15] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB*, pages 251–262, 1996.

[16] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1):647–659, 2008.

[17] A. Polleres. From SPARQL to rules (and back). In *WWW*, pages 787–796, 2007.

[18] R. Pottinger and A. Y. Levy. A scalable algorithm for answering queries using views. In *VLDB*, pages 484–495, 2000.

[19] K. Q. Pu, V. Hristidis, and N. Koudas. Syntactic rule based approach to Web service composition. In *ICDE*, 2006.

[20] A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns. In *PODS*, pages 105–112, 1995.

[21] P. Senellart, A. Mittal, D. Muschick, R. Gilleron, and M. Tommasi. Automatic wrapper induction from hidden-Web sources with domain knowledge. In *WIDM*, 2008.

[22] D. E. Simmen, M. Altinel, V. Markl, S. Padmanabhan, and A. Singh. Damia: data mashups for intranet applications. In *SIGMOD*, 2008.

[23] F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: A Large Ontology from Wikipedia and WordNet. *Elsevier Journal of Web Semantics*, 2008.

[24] M. Technologies. The freebase project. `http://freebase.com`.

[25] V. Vassalos and Y. Papakonstantinou. Describing and using query capabilities of heterogeneous sources. In *VLDB*, pages 256–265, 1997.

[26] Word Wide Web Consortium. RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation 2004-02-10.

[27] Word Wide Web Consortium. XSL Transformations (XSLT). W3C Recommendation 1999-11-16.

[28] World Wide Web Consortium. SPARQL Query Language for RDF (W3C Recommendation 2008-01-15), 2008. `http://www.w3.org/TR/rdf-sparql-query/`.

[29] F. Wu and D. S. Weld. Automatically refining the Wikipedia infobox ontology. In *Proc. of the Int. WWW Conf.*, pages 635–644, New York, NY, USA, 2008. ACM.

[30] C. Yu and L. Popa. Constraint-based XML query rewriting for data integration. In *SIGMOD Conference*, pages 371–382, 2004.