ANGIE: Active Knowledge for Interactive Exploration

Nicoleta Preda Fabian M. Suchanek Gjergji Kasneci
Thomas Neumann Maya Ramanath Gerhard Weikum
MaxPlanck Institute for Informatics
{npreda,suchanek,kasneci,neumann,ramanath,weikum}@mpi-inf.mpg.de

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. This information is integrated seamlessly and transparently into the local database, so that the user can query and browse the knowledge base while appropriate Web services are called automatically in the background.

1. INTRODUCTION

Recent advances on automated information extraction [6] 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 [19, 10], Freebase [20], KnowItAll [4], or Intelligence-in-Wikipedia [25] 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, 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. The current implementation uses YAGO [19] 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 combine data from local and dynamic sources in an arbitrary manner, without having to know where the data actually resides or is constructed.

Related Work. Several approaches have combined data from different sources: XML and Web services[1], RSS and Web ser-

VLDB '09, August 24-28, 2009, Lyon, France

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

vices[18, 9], semantic databases[5], Web forms and Web services [17]). However, the interaction of a semantic knowledge base with dynamic Web services has not been studied before. Our work draws inspiration from Web service composition [15], entity resolution on the fly [18], the relationship between SPARQL and Datalog [13], and schema mappings [2].

Several works [11, 14, 7, 16] proposed to use views for answering queries. Different from views, Web services deliver incomplete information, require certain pre-conditions and cannot be fully materialized. [21] considered a different scenario where the source itself has specific querying capabilities. With such a functional view of the data, query translation needs to observe the specific constraints of a source's interface. There is ample work on data integration systems, but most prior work used an object oriented representation (e.g., [8]) or XML [26], which are both less adequate for the RDF knowledge bases that we consider.

2. FRAMEWORK

2.1 RDF graphs

In tune with recent work [19, 3, 10], we represent our knowledge base as an RDF graph [22]. An RDF graph is a directed labeled graph, in which the nodes are entities (such as individuals and literals) and the labeled edges represent relationships between the entities. A fragment of a sample RDF graph is shown in Figure 1. We refer to an edge and its two adjacent nodes also as a fact or an RDF statement (aka. subject-property-object triple).

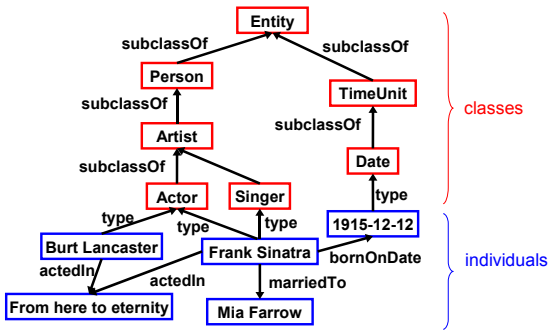

Figure 1: An excerpt of an RDF graph

RDFS allows *reifing* a statement [22]. This means that a statement can in turn be an entity in the RDF graph. This allows other statements to have that statement as an argument, and is convenient for representing ternary relations like events or temporal or provenance annotations of facts.

Figure 2: Two sample queries

For querying, we consider a subset of the SPARQL standard query language [24]. We can view a query as a small RDF graph (template) that can have variables as node or edge labels. Figure 2 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 Mia Farrow.

An answer for a query is a subgraph of the knowledge base that matches the query. For instance, consider again the RDF graph shown in Figure 1. The query Q_2 has an answer in the RDF graph because there is the substitution $\sigma(\$r)$=marriedTo. $\sigma(Q_2)$=*(FrankSinatra, marriedTo, AvaGarner)* is a sub-graph of the RDF graph. There is, however, no answer to Q_1 in the current database. Therefore, a Web service has to be called.

2.2 Functions

In the ANGIE framework, we view a Web service as a function, which, given input parameters, returns output values. We model a function definition also as an RDF graph that can contain variables like a query. The edges of the function definition are partitioned into *pre-conditions* and *post-conditions*. Figure 3 shows a sample function definition with the pre-conditions in blue and the post-conditions in red.

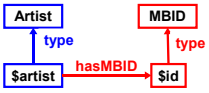

Figure 3: A sample function definition

Intuitively speaking, the pre-condition edges have to be fulfilled before the function can be called. In the example, an instance of the class *artist* has to be provided. The post-conditions specify the results of the function. In the example, a new node is added to the knowledge graph that represents the id of the artist in the MusicBrainz world. The function tells us that the artist and the id are linked by an edge *hasMBID* and that the id is a MusicBrainz id.

In our setting, the function definitions are given. We do not consider retrieving them automatically. Rather, we assume that the appropriate WSDL or REST interfaces have already been identified and cast into functions represented by RDF graphs. Technically, the pre-conditions and the post-conditions of a function definition are RDF statements (with variables). Using reification, these statements can be represented in the knowledge base. This way, function definitions become first-class citizens of the knowledge base.

3. QUERY PROCESSING

Given a query that needs to be resolved by a function call to a Web service, the ideal case would be to map bind the query's output variable (a node of a post-condition edge) to a single function provided by the service. However, as pointed out earlier, services or the user's queries are often not that simple, and instead we may have invoke multiple functions in an appropriately composed manner. Our goal then is to combine the function definitions into a valid sequence of function calls to answer the query. Formally, a family of calls to a function with certain parameters is defined as a *function instantiation*: a function instantiation for a function definition f is a substitution σ for the variables of the function definition. Some variables can be mapped to constants. Other variables are simply renamed by σ. This is necessary, because f may be instantiated multiple times and naming conflicts have to be avoided. Intuitively speaking, a function instantiation creates a copy of the function definition in which all variables have been replaced.

We combine function instantiations into *function compositions*. A function composition 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 $f_1, ..., f_n \in F$ on a common set of fresh variable names, so that (1) for each pre-condition edge pre of any function f_i,

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

and (2)

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

A function composition is a set of function instantiations such that each edge in the query and each pre-condition of a function instantiation is satisfied by the post-conditions of another function instantiation.

For uniformity, let us also consider the triples in the database as if they were 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 post-condition edge in which all three components are variables. Whenever this function is "called", it issues a query to the local RDF knowledge base. One query can be answered by several function compositions. Therefore, we translate a query into a (possibly infinite) sequence of function compositions. This way we can incrementally retrieve a larger set of results, as some results are returned only be particular function compositions. For example, a function composition based on an actor's name or id may retrieve only a partial list of the actor's albums (e.g., because of service restrictions on result sizes). So one could identify, in the local knowledge base, other actors or directors that the actor of interest has often worked with, retrieve their movies and those movies' actors by a different function composition, and test the results whether for the originally given actor. Alternatively, multiple services each with incomplete but complementary data could be invoked in a similar manner, again to obtain higher recall.

Algorithm 1 translate(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: translate($F, Q \cup \sigma''(pre(f)) \setminus \{q\}, R \cup \{(f, \sigma'')\}$)
12: **end if**
13: **end for**
14: **end for**

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

In Line 6, the algorithm tries all available function definitions. We are currently developing an estimation for the cost of a function call. This way, the algorithm can prioritize functions that are expected to yield results faster or with lower execution costs. In our current implementation, the database function f_{db} is always preferred over other functions. This way, the algorithm outputs a (possibly infinite) sequence of function compositions, starting with the most promising compositions. Each function composition will be evaluated and will return a set of results. We set a limit on the maximum number of function calls.

4. SYSTEM IMPLEMENTATION

The overall architecture of the ANGIE system is illustrated in Figure 4. The system uses the existing YAGO ontology [19], 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].

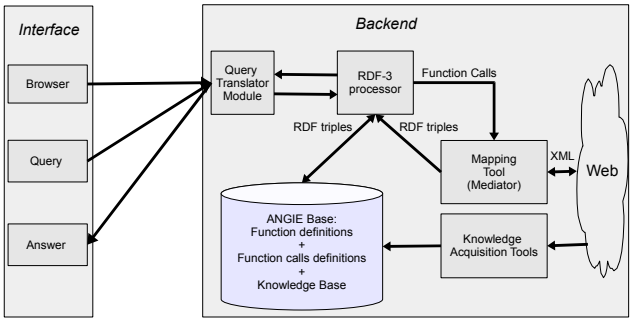

Figure 4: System architecture of ANGIE

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 [12]. 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[23] for this purpose. The entities can then be handled by the RDF-3X processor.

User Interface. The user interface allows the user to query the knowledge base in the language described in Section 2.1. 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.

5. DEMO SCENARIO

ANGIE is a light-weight desktop tool for querying, browsing, and visualizing the active knowledge base. Expert users can query the knowledge base using complex queries. Since the answers to the queries are graphs, they can be visualized in ANGIE. Casual users can browse a hyperbolic visualization of the complete knowledge base. Starting from one node in the graph, the user can explore further entities. For example, when the user clicks on the Frank Sinatra node, all facts about Frank Sinatra are displayed. At the same time, Web service function compositions are called in the background to retrieve more information about the singer. In Figure 5, the albums of Frank Sinatra (marked here in green for illustration purposes) have been retrieved on demand from the MusicBrainz Web service. Analogously, when the user clicks on Mia Farrow, facts about her movies are retrieved from a service like InternetVideoArchive. If appropriate services were available, one could even retrieve the music featured in Mia Farrow's movies and check if it includes a Sinatra song.

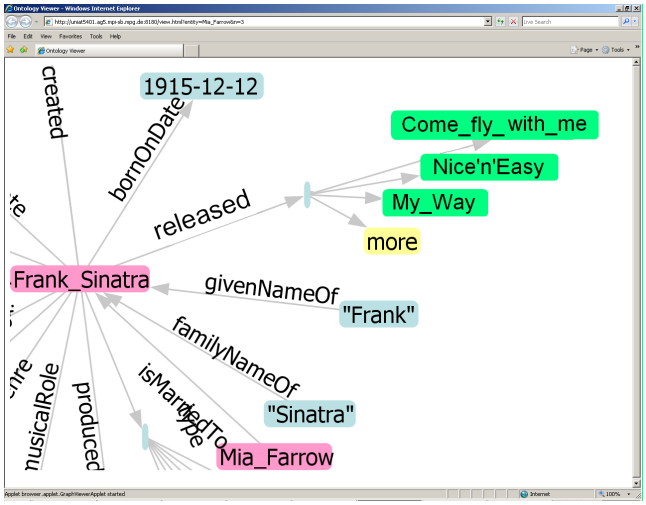

Figure 5: Screenshot of ANGIE after calling service functions

This is one possible scenario that demo visitors can try online. ANGIE supports seamless connections with various other services for music, books, and movies. Thus, many other scenarios are possible to interactively explore the active knowledge base and discover interesting facts about favorite musicians, writers, actors, directors, songs, movies, and their cross-relationships.

6. REFERENCES

[1] Serge Abiteboul, Omar Benjelloun, and Tova Milo. The Active XML project: an overview. *VLDB J.*, 2007.

[2] Bernd Amann, Irini Fundulaki, Michel Scholl, Catriel Beeri, and Anne-Marie Vercoustre. Mapping XML fragments to community Web ontologies. In *WebDB*, 2001.

[3] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. DBpedia: A nucleus for a Web of Open Data. *The Semantic Web*, 2008.

[4] Michele Banko, Michael J. Cafarella, Stephen Soderland, Matthew Broadhead, and Oren Etzioni. In Manuela M. Veloso and Manuela M. Veloso, editors, *IJCAI*.

[5] Christian Bizer, Tom Heath, Kingsley Idehen, and Tim Berners-Lee. Linked data on the web (LDOW2008). In *WWW*, 2008.

[6] AnHai Doan, Luis Gravano, Raghu Ramakrishnan, and Shivkumar Vaithyanathan (Eds.). Special issue on managing information extraction. *ACM SIGMOD Record 37(4)*, 2008.

[7] Oliver M. Duschka, Michael R. Genesereth, and Alon Y. Levy. Recursive query plans for data integration. *J. Log. Program.*, 43(1), 2000.

[8] Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey D. Ullman, Vasilis Vassalos, and Jennifer Widom. The tsimmis approach to mediation: Data models and languages. *J. Intell. Inf. Syst.*, 8(2), 1997.

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

[10] Gjergji Kasneci, Fabian M. Suchanek, Georgiana Ifrim, Maya Ramanath, and Gerhard Weikum. NAGA: Searching and Ranking Knowledge. In *ICDE*, 2008.

[11] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB*, 1996.

[12] Thomas Neumann and Gerhard Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1), 2008.

[13] Axel Polleres. From sparql to rules (and back). In *WWW*, 2007.

[14] Rachel Pottinger and Alon Y. Levy. A scalable algorithm for answering queries using views. In *VLDB*, 2000.

[15] Ken Q. Pu, Vagelis Hristidis, and Nick Koudas. Syntactic rule based approach to Web service composition. In *ICDE*, 2006.

[16] Anand Rajaraman, Yehoshua Sagiv, and Jeffrey D. Ullman. Answering queries using templates with binding patterns. In *PODS*, 1995.

[17] Pierre Senellart, Avin Mittal, Daniel Muschick, Rémi Gilleron, and Marc Tommasi. Automatic wrapper induction from hidden-Web sources with domain knowledge. In *WIDM*, 2008.

[18] David E. Simmen, Mehmet Altinel, Volker Markl, Sriram Padmanabhan, and Ashutosh Singh. Damia: data mashups for intranet applications. In *SIGMOD*, 2008.

[19] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. YAGO: A Large Ontology from Wikipedia and WordNet. *Elsevier Journal of Web Semantics*, 2008.

[20] Metaweb Technologies. The freebase project. `http://freebase.com`.

[21] Vasilis Vassalos and Yannis Papakonstantinou. Describing and using query capabilities of heterogeneous sources. In *VLDB*, 1997.

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

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

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

[25] Fei Wu and Daniel S. Weld. Automatically refining the Wikipedia infobox ontology. In *Proc. of the Int. WWW Conf.*, 2008.

[26] Cong Yu and Lucian Popa. Constraint-based xml query rewriting for data integration. In *SIGMOD Conference*, 2004.