NGS: A New Generation Search Engine Supporting Cross Domain Queries* (Extended Abstract)

Daniele Braga[1], Diego Calvanese[2], Alessandro Campi[1], Stefano Ceri[1], Florian Daniel[1], Davide Martinenghi[1], Paolo Merialdo[3], Riccardo Torlone[3]

[1] Dip. di Elettronica e Informazione, Politecnico di Milano, Milano, Italy
[2] Faculty of Computer Science, Free University of Bozen-Bolzano, Bolzano, Italy
[3] Dip. di Informatica e Automazione, Università Roma Tre, Roma, Italy

Abstract. This paper presents NGS, a framework providing fully automated support for cross-domain queries. In particular, NGS (a) integrates different kinds of services (search engines, web services, and wrapped web pages) into a global ontology, i.e., a unified view of the concepts supported by the available services, (b) covers query formulation aspects over the global ontology, and query rewriting in terms of the actual services, and (c) offers several optimization opportunities leveraging the characteristics of the different services at hand, based on several different cost metrics.

1 Introduction and motivation

While an increasing amount of search services on the Web becomes available, they still work in isolation; their intrinsic limit is the inability to support complex queries ranging over multiple domains. Answering a query such as "find all database conferences held within six months in locations whose seasonal average temperature is 28°C and for which a cheap travel solution exists" requires combining search engines specialized over different domains. For instance: (i) finding interesting conferences in the desired timeframe on online services made available by the given scientific community; (ii) finding if the conference location is served by low-cost flights; (iii) finding if there are luxury and cheap hotels in proximity of the conference location.

This paper describes a framework for the development of New Generation Search (NGS) supporting queries over multiple, specialized search engines, developed in the context of a project funded by the Italian Government. Our framework makes use of service-enabled and XML-related technologies, and of ontological knowledge in the context of data mapping. In NGS we distinguish between exact services and search services. *Exact* services have a "relational" behavior and return either a single answer or a set of answers which are not ranked. *Search* services return a list of answers in ranking order, according to

* Supported by Italian PRIN project "New technologies and tools for the integration of Web search services".

some measure of relevance; such measure may be either visible in the result or opaque. Services returning many answers have an associated *selectivity*, expressing the average size of the result. They can further be classified as "chunked" or "bulk"; in the former case, they return results in chunks of a fixed size, whereas in the latter case they return their result set as a whole.

The main contributions of this paper are: (a) a multi-level model for expressing queries over web services and search engines – this model covers a conceptual level, where queries are expressed as conjunctive expressions over arbitrary predicates; a logical level, where queries are mapped to services; and a physical level, where queries are expressed as execution strategies over services, with given methods for service invocation and for search engine integration; (b) several strategies for performing the transformations required by these models, and in particular for mapping a conceptual query into several logical queries (adapting well-known mapping techniques) and for optimizing the logical queries, thus producing the best execution strategy – optimization requires the definition of several, alternative metrics; (c) the inclusion within this framework of additional steps, such as query augmentation (how to extend a query when it cannot immediately be mapped to a service) and source wrapping (how to build wrappers over data sources offering Web service interfaces); (d) an architecture for implementing the framework, supporting service registration and offering query interfaces for end-user interaction.

2 Overview of the approach

2.1 Running example

We consider as a running example the query reported in the Introduction: "find all database conferences held within six months in locations whose seasonal average temperature is 28 degrees and for which a cheap travel solution exists". We assume that queries can be expressed over a conceptual schema, represented in Figure 1, consisting of 3 relations: `travel`, describing the details of flights and hotels being selected; `climate`, describing weather conditions expected at given dates in given locations; and `conference`, describing the conference offerings in given subjects.

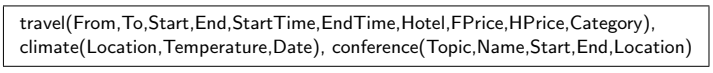

Fig. 1. Schema of conceptual services derived from the global ontology

The running query can be expressed by the Datalog expression in Figure 2, where the terms preceded by a $ sign are user input parameters. Datalog notation is chosen for its elegance and simplicity, but we currently focus on conjunctive queries (i.e., select-project-join queries) without recursion. Note that "cheap solution" is translated as a predicate over the overall cost of the solution; thus,

```
q(Conf,City,HPrice,FPrice,Start,StartTime,End,EndTime,Hotel) ←
  travel($from, City, Start, End, StartTime, EndTime, Hotel, FPrice, HPrice, $category),
  climate(City, Temperature, Start), conference('DB', Conf, Start, End, City),
  Start≥$startDate, End≤$startDate+180, temperature≥28, FPrice+HPrice<2000.
```

Fig. 2. Query over the conceptual services

the problem considered in this paper could be further expanded into the use of domain knowledge for query interpretation. Such expansion is indeed feasible because we assume a complete knowledge of the semantic domains of Web services, and is planned within our project.

The query can be answered by six physical services which have been previously registered, represented in Fig. 3. These are: two services for conference offerings, two services for hotel offerings, and one service for flight offerings and for weather conditions. The presence of many physical services is due to the fact that the same information may have some access limitations, i.e., be accessed according to different access patterns. For instance, conferences may be queried by setting the conference's topic, or by setting the conference's location. Thus, services are denoted not only by the parameters that they expose to queries, but also by the role of the parameters (input vs output), and therefore the representation of a service is that of an adorned Datalog predicate, where places are either bound or free; in Fig. 3, bound places are in boldface. We denote services over the same data but with different adornments by a different index.

confSchedule$_{(1)}$(**Topic**, Name, Start, End, City)
confSchedule$_{(2)}$(Topic, Name, Start, End, **City**)
weather(**City**, Temperature, **Date**)
flight^{S}(**From**, **To**, **OutDate**, **RetDate**, OutTime, RetTime, Price)
hotel$^{S}_{(1)}$(Name, **City**, **Category**, **CheckInDate**, **CheckOutDate**, Price)
hotel$^{S}_{(2)}$(Name, City, Category, CheckInDate, CheckOutDate, Price)

Fig. 3. Services at the physical level

A fundamental distinction in our model concerns the nature of services. *Exact services* have a standard relational behavior and return either a single answer or a set of answers which are not ranked. Conversely, *search services* return answers in relevance order: their management within a query requires special care, because in general the answers to a search service are very numerous, but users are only concerned with the first answers. Thus, expanding a query to incorporate all the results of a search service would be wrong. Moreover, the user expects results in ranking order; thus, by composing answers from multiple services, we must produce a global ranking that is a good composition of the various partial rankings. We denote search services by giving them the "S" superscript. For example, `flight` is a search service: it requires the origin and destination locations as well as the departure and return dates, and outputs a list of flight solutions, including their times and prices, in increasing price order.

2.2 System architecture

Our envisioned framework consists of three layers:

- ***Query formulation layer***. First, this layer allows users to specify their requests to the NGS system by using an interface which refers to concepts of the global ontology. The query language and the ontology hide the specificity of the services as implemented and available online. The main role of this layer is to rewrite the user query into a logical expression of Web Service calls. Queries are rewritten through mappings, and the result of this rewriting is expressed in terms of Datalog programs in the form of multi-domain conjunctive queries over physical services data with access limitations; when access limitations are too strict and prevent from reaching any answer, query expansion mechanisms can be also used. Note that, in general, the availability of different access patterns for the same service may give rise to several alternative rewritings of the query. The issues concerning the query formulation layer are described in Sections 3.1.
- ***Query execution layer***. This layer receives the Datalog programs generated by the previous level. The role of this layer is to generate a query plan optimized taking into account the parameters associated to the services and the cost model. This optimization is done taking into account several aspects, such as: (i) the types of operations involved in the query plan; (ii) available profiling information on specific services; (iii) ranking of the results. The issues concerning the query execution layer are described in Section 3.2.
- ***Data layer*** The data layer addresses the representation in the framework of the physical services; they may be either Web Services or wrapped, data-intensive Web sites. Services are constantly profiled so as to feed the optimizer of the layer above with estimates of the figures which are relevant to the optimization problem (such as response time, average number of returned results, statistical distribution of values into the results, and typical decrease trend in the function form of the relevance). When information sources are wrapped, we envision resorting to automatic wrapper generation techniques, so as to easily and readily maintain the wrappers aligned with the evolution of the Web Sites. The issues concerning the wrapping of Web sources are described in Section 3.3.

3 The main components of the architecture

3.1 Query mapping

Our considerations on the mapping between the global ontology and the service schemas are drawn from work in data integration, where two basic approaches have been proposed to specify the mapping between a global ontology (or global schema, in data integration terminology) and a set of services (or data sources) [1]: *global-as-view* (or simply GAV), which requires that the global ontology is expressed in terms of the services' schemata, and *local-as-view*

```
travel(From, To, Start, End, StartTime, EndTime, Hotel, FPrice, HPrice, Category) ←
  flight(From, To, Start, End, StartTime, EndTime, FPrice),
  hotel(Hotel, To, Category, Start, End, HPrice).
```

Fig. 4. Example of GAV mapping for the `travel` service

```
q(Conf, City, HPrice, FPrice, Start, StartTime, End, EndTime, Hotel) ←
  flight($from, City, Start, End, StartTime, EndTime, FPrice),
  hotel(Hotel, City, $category, Start, End, HPrice),
  confSchedule('DB', Conf, Start, End, City), weather(City, Temperature, Start),
  Start≥$startDate, End≤$startDate+180, temperature≥28, FPrice+HPrice<2000.
```

Fig. 5. Logical query over the available services

(LAV), which requires the global ontology to be specified independently from the services. Intuitively, the GAV approach provides a method for specifying the integration system with a more procedural flavor with respect to the LAV approach. Indeed, whereas in LAV the designer of the system may concentrate on specifying the content of the services in terms of the global ontology, in GAV the burden of specifying how to get the data of the global ontology by queries over the services is entirely on the designer.

The approach taken in NGS is one where new services are registered in the system by mapping their information content to the terms of the global ontology. To simplify for the designer the task of specifying such mappings, we envision to follow a LAV approach. However, to allow for query processing by unfolding, as in GAV, we intend to adopt the techniques proposed in [2] to convert a LAV system into an equivalent GAV system by introducing suitable constraints (essentially, inclusion dependencies) in the global ontology.

An example of a suitable GAV mapping for `travel` is shown in Figure 4. According to this and similar mappings for the other entities and services, the query over the global ontology can be rewritten, in general, as a union of conjunctive queries over the available services. In our running example we obtain the query shown in Figure 5.

In the context of query answering over Web Services, queries can be conceived as in the traditional relational setting, but with the extra requirement that certain fields be mandatorily filled in by the user in order to obtain a result. As mentioned, we assume that each service at the physical level is equipped with an adornment specifying its input parameters, called *access pattern*, as shown in Figure 3, where input fields are marked in boldface. Any query formulated over such services needs to comply with the physically available access patterns. For this reason, a conjunctive query, such as the one of Figure 5, where the order of the literals in the body is immaterial, needs to be further instantiated into what we call a *logical access plan*. First of all, for each service with more than one adornment, one of the available access patterns has to be chosen. Besides, an order of "execution" of the literals in the body of the query has to be determined so that all the access patterns of all invoked services are respected, i.e., for each input argument there is either a value provided directly by the user, or a binding

```
q(Conf, City, HPrice, FPrice, Start, StartTime, End, EndTime, Hotel) ←
  confSchedule(1)('DB', Conf, Start, End, City), Start≥$startDate,
  End≤$startDate+180, weather(City,Temperature,Start), Temperature≥28,
  flight($from, City, Start, End, StartTime, EndTime, FPrice),
  hotel(1)(Hotel, City, $category, Start, End, HPrice), FPrice+HPrice<2000.
```

Fig. 6. Logical access plan for the query of Fig. 5 using `confSchedule` first

```
q(Conf, City, HPrice, FPrice, Start, StartTime, End, EndTime, Hotel) ←
  hotel(2)(Hotel, City, $category, Start, End, HPrice), Start≥$startDate,
  End≤$startDate + 180, confSchedule(2)('DB', ConfName, Start, End, City),
  weather(City, Temperature, Start), Temperature≥28,
  flight($from, City, Start, End, StartTime, EndTime, FPrice), FPrice+HPrice<2000.
```

Fig. 7. Logical access plan for the query of Fig. 5 using `hotel` first

is available from an output field of a previously invoked service. We still write a logical access plan as a conjunctive query, and make it clear what access pattern is used for each service by indicating in subscript the corresponding index. As is customary, we will use the left-to-right order of appearance of the literals in the query body to indicate a class of possible invocation orders, meaning that if a service s_1 occurs left of service s_2 in the query, then s_1 is not invoked after s_2.

A query under access limitations is said to be *feasible* if there exists an equivalent query that is executable as is from left to right, while respecting the access limitations. Whenever the query is feasible but admits several logical access plans, they are all given to the logical layer, which will then select the most promising ones according to some cost metric. In our running example, the logical query of Figure 5 admits several logical access plans, due to the fact that multiple access patterns are available for several services. Two possible plans for this query are shown in Figures 6 and 7.

3.2 Query optimization

Starting from the set of alternative logical access plans, the query execution layer is in charge of (i) deriving one or more executable *physical access plans* for each of the alternative logical access plans and according to a given optimization strategy and (ii) identifying the *best* physical execution plan according to a given cost metric. The set of alternative logical access plans considered at this stage contains only plans that are *executable* according to their access limitations; non-executable plans have been discarded in the previous step.

If we consider the logical access plans shown in Figures 6 and 7, we can for instance derive the physical access plans of Figure 8 and 9, resp.

The plans make use of a graphical modeling notation that we use to represent a physical access plan. In this notation *selective* exact services (that is, returning at most one tuple) are represented as simple boxes and *proliferative* exact services (that is, returning on the average more than one tuple) are represented as boxes labeled with a "*". Search services are represented as boxes with a grey trapezium (sketchily representing the decrease in ranking of the results). If a

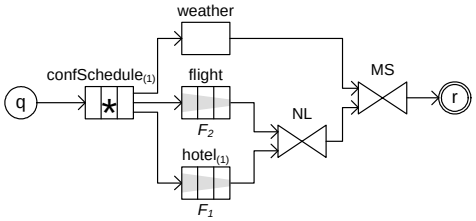

Fig. 8. Possible physical access plan for the logical access plan of Fig. 6.

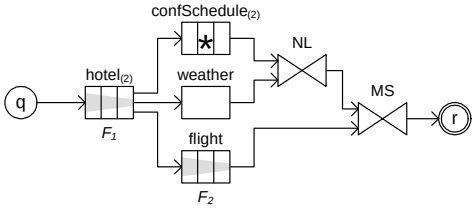

Fig. 9. Possible physical access plan for the logical access plan of Fig. 7.

service supports the chunking of its output into smaller fragments, we show that a particular access plan makes use of the service's chunking feature by splitting the service's box into three smaller boxes. We distinguish between two join patterns: *parallel join* and *pipe join*. The parallel join is represented by means of a dedicated join symbol with an associated label ("NL" or "MS") expressing the respective join strategy. The pipe join is denoted by an arrow connecting two nodes, indicating that the join is computed by feeding with the output of the origin the input of the destination. Finally, an access plan has a unique start node (the user query's input) and a unique end node (the query result).

Once all logical access plans have been expanded into their candidate physical execution plans according to the given optimization strategy, the identification of the best physical execution plan is based on a suitable *cost metric*, which allows us to associate a cost estimation to each physical execution plan.

3.3 Source wrapping

One of the novelties of our approach is the involvement of Web Services specialized in the extraction of contents from data-intensive Web sites (e.g., wrappers of sites exposing bond quotes or the personnel of a given research institute). In order to develop a scalable system, it is recommended that the generation of wrappers is performed as automatically as possible. Several approaches have been proposed for the automatic generation of wrappers (see [4] for a recent survey).

In data-intensive sites, pages are usually automatically generated using scripts which extract the content of the database, first executing some queries and then

serializing the extracted dataset into HTML code. A nice property of these sites is that pages generated by the same script share a common structure. We call a *class* a collection of pages in a site generated by the same script. We may then re-formulate the problem as follows: "given a set of sample HTML pages belonging to the same class, find a data structure capable to carry the same information of the original Web pages".

Arasu and Garcia-Molina proposed ExAlg [5], an algorithm for extracting structured data from Web pages generated by encoding data from a database into a common template. To discover the template (i.e., characterizing the class of pages), ExAlg uses so called Large and Frequently occurring EQuivalent classes (LFEQ), i.e. sets of words that have similar occurrence pattern in the input pages. Conversely, RoadRunner [6] abstracts a wrapper as a regular grammar, whose productions are inferred and refined by iteratively parsing the input pages. Roadrunner leverages page regularity by exploiting similarities and differences among pages by means of *Match*, an unsupervised algorithm that iteratively refines a wrapper, by iteratively parsing pages of the sample set and generalizing the wrapper whenever the parsing process fails.

The two approaches described last have complementary strengths and limitations. In NGS, we have developed a combined technique that aims at conciliating them, thus overcoming their limitations while leveraging their strengths. Based on the formal background of RoadRunner, we have introduced a preprocessing phase, which enriches the input pages by means of information derived from a statistical analysis inspired by that proposed in ExAlg. The goal of the transformation is to remove ambiguities that sometimes keep the RoadRunner algorithm from inferring the grammar. Also, a post processing phase is run over the extracted data in order to detect disjunctive patterns. Whenever these are found for each branch of the disjunction, a wrapper is recursively inferred.

References

1. J. D. Ullman, "Information integration using logical views," in *Proc. of ICDT'97*, ser. LNCS, vol. 1186. Springer, 1997, pp. 19–40.
2. A. Calì, D. Calvanese, G. De Giacomo, and M. Lenzerini, "On the expressive power of data integration systems," in *Proc. of ER 2002*, 2002.
3. D. Braga, A. Campi, S. Ceri, and A. Raffio, "Joining the results of heterogeneous search engines," *Information Systems*, submitted to.
4. M. Kayed and K. F. Shaalan, "A survey of web information extraction systems," *IEEE Trans. on Knowledge and Data Engineering*, vol. 18, no. 10, pp. 1411–1428, 2006.
5. A. Arasu, H. Garcia-Molina, and S. University, "Extracting structured data from web pages," in *Proc. of ACM SIGMOD*, 2003, pp. 337–348.
6. V. Crescenzi, G. Mecca, and P. Merialdo, "Roadrunner: Towards automatic data extraction from large web sites," in *Proc. of VLDB 2001.* San Francisco, CA, USA: Morgan Kaufmann, 2001, pp. 109–118.