NGS: a Framework for Multi-Domain Query Answering

D. Braga*, D. Calvanese†, A. Campi*, S. Ceri*, F. Daniel*, D. Martinenghi*, P. Merialdo‡, R. Torlone‡

*Dip. di Elettronica e Informazione, Politecnico di Milano, Piazza L. da Vinci 32, Milano, Italy
{braga,campi,ceri,daniel,martinen} at elet.polimi.it

†Faculty of Computer Science, Free University of Bozen-Bolzano, Bolzano, Italy
calvanese at inf.unibz.it

‡Dip. di Informatica e Automazione, Università di Roma Tre, Via della Vasca Navale 79, Roma, Italy
{torlone,merialdo} at dia.uniroma3.it

***Abstract*— If we consider a query involving multiple domains, 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", we note that (i) *general-purpose* search engines fail to answer multi-domain queries and (ii) *specific* search services may cover one of such domains, but no general integration framework is readily available. Currently, the only way to treat such cases is to separately query dedicated services and feed the result of one search as input to another, or to pairwise compare them by hand.**

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.

I. Introduction and motivation

The current evolution of the Web is characterized by an increasing number of search engines and query interfaces, ranging from generic ones (Google) to domain-specific ones (geo-localization services or on-line catalogs). Meanwhile, wrapping technology is evolving so as to enable the development of specialized services extracting content from data-intensive Web sites (e.g., wrappers of sites delivering bond quotes), and exposing them as Web Services.

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 the query reported in our abstract 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 by three University groups 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 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 queries 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.

A. Running example

We consider as a running example the query reported in the abstract: "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".

```
travel(From, To, Start, End, StartTime, EndTime, Hotel, FPrice,
      HPrice, Category)
climate(Location, Temperature, Date)
conference(Topic, Name, Start, End, Location)
```

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

```
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

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. Thus, the running query is expressed as a 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, 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.

We next move to Web service descriptions. 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; hotels may be queried by setting a complete information (hotel and period identification) or by giving no information at all (and then searching particularly good special offers). Moreover, services may themselves hide other services (e.g., a search engine integrator for flights) or instead be offered by an individual organization (e.g., Hyatt Regency's hotel offerings). 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.

A fundamental distinction in our model concerns the nature of services. Search services return answers in relevance order. Their management within a query requires special care, because in general the answers to a search services are very

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

Fig. 3. Services at the physical level

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. On the other hand, the user is the only one that can correctly evaluate the relevance of answers produced by search engines; therefore, answers of search engines should be composed in the query's output and presented to the user for a correct evaluation. 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, and use the global ranking in producing the output; then, the user will decide at each interaction how many results should be considered (e.g., before halting the query and re-issuing another query with different parameters). 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. The `hotel` service is also a search service, available in two versions: the first one requires a city, a category, a check-in date, and a check-out date, and returns names and prices of hotel accommodations matching the input parameters; we can think this to be offered by a booking system acting as an integrator of other services. The second version is invoked without any bound parameter; we can think this to be offered by a chain of hotels, e.g. Hyatt Regency, returning only the best offers in the world according to an internal ranking order that corresponds to the "best" offer from the chain's perspective.

B. *System architecture*

Our envisioned framework is summarized by the architectural view of Figure 4. The user poses a query over the global ontology, which is equipped with a set of mappings with the services' schemata, and possibly some integrity constraints [1]. The query is rewritten according to the mappings and the constraints as a query over the services. This is then transformed into several possible executable query plans taking into account possible limitations in accessing the services. Among data services we list content extracted via wrappers from data-intensive Web sites.

More in detail, the 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.

[1]The ontology is the result of a data integration process, outside of the scope of this paper; we start from its relational formulation, given in Fig. 1.

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 II and III.

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 IV.

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, 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 V.

In the rest of the paper we describe each of the framework layers; we conclude the paper with a description of the forthcoming development of the project.

II. QUERY MAPPING

We now discuss the aspects related to the specification of the mapping between the global ontology and the service schemas, and the use of such a mapping in the query answering process. Our considerations 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], [2], [3].

A. Global-as-view approach

The first approach, called *global-as-view* (or simply GAV), requires that the global ontology is expressed in terms of the services' schemata. More precisely, to every element of the global ontology, a view (i.e., a query) over the services is associated, so that its meaning is specified in terms of the data retrieved from the services.

An advantage of the GAV approach is that query processing is generally easy, since we can take advantage of the fact that

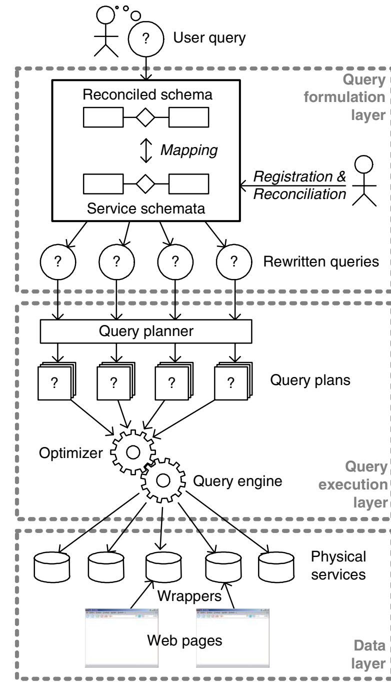

Fig. 4. Reference scenario

the mapping directly specifies, in terms of the views over the services, which services contribute data to which element of the global schema. Hence, in a (pure) GAV system, a simple unfolding strategy is sufficient to take into the mapping during query answering. On the other hand, a system based on GAV is rather difficult to maintain, since adding a new service may in principle affect the queries associated to all concepts in the global ontology.

B. Local-as-view approach

The second approach, called *local-as-view* (LAV), requires the global ontology to be specified independently from the services. In turn, the information content of each service is defined as a view over the global ontology.

The advantage of the LAV approach is that it ensures an easier extensibility of the system, and provides a more appropriate setting for its maintenance. Indeed modifying a service or adding a new one to the system requires only to modify or provide the definition of the added service, and does in general not involve further changes in the global ontology. On the other hand, processing queries in the LAV approach is a difficult task [2], [4], [5], [3]. Indeed, the only knowledge we have about the data in the global ontology is through the views representing the services, and such views provide only partial information about the data. Since the mapping associates to each service a view over the global ontology, it is not immediate to infer how to use the services in order to answer queries expressed over the global ontology. Thus, extracting information from the system is similar to query answering in the presence of incomplete information, which is a complex task [6], [7].

```
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. 5. 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. 6. Logical query over the available services

C. Comparison between LAV and GAV and approach of NGS

A comparison of the LAV and the GAV approaches is reported in [2], [8]. 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 [8] to convert a LAV system into an equivalent GAV system by introducing suitable constraints (essentially, inclusion dependencies) in the global ontology. Such dependencies, together with those already present in the ontology establishing the taxonomic relationship between classes, need then to be taken into account during query processing. As shown by recent work on query answering under constraints [9], [10], [11], this can done by an initial expansion step on the user query that precedes unfolding due to the mappings. A crucial aspect is that these transformations on the user query, which essentially compile into the resulting query both the ontology constraints and the mappings, can be carried out without the need to access the data provided by the services [10]. This holds for a rather wide class of ontology languages [12], comprising those that we envision to use in NGS, and that essentially corresponds to a taxonomy of classes and data types.

An example of a suitable GAV mapping for `travel` is shown in Figure 5. 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 6.

III. Access limitations and query expansion

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 1, where input fields are marked in boldface. Any query formulated over such services needs to comply with the physically available access patterns, as discussed in [13], [14]. For this reason, a conjunctive query, such as the one of Figure 6, 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 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 the 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.

According to the available access patterns for the services, some logical access plan complying with the access patterns of all services used in the query may or may not exist. We distinguish these two cases in the following.

A. Feasible queries

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 admits exactly one feasible rewriting, there is exactly one possible logical access plan; this is then directly passed onto the logical layer in NGS for further choices on the execution at the physical level. Conversely, if several logical access plans exist for the query, they are all given to the logical layer, which will then select the most promising ones according to some cost metric.

The problem of determining feasibility is analyzed, e.g., in [15], [16], where, in particular, it is studied whether there is an ordering of subgoals that enables answering the query, and, if multiple such orderings are possible, how to pick the best ordering according to the principle that "bound is better". Typically, a service adornment with a superset of the input fields of another adornment will be cheaper to use in terms of execution time, since its result space is somehow more restricted. However, we do not want to limit ourselves to this cost metric, and, indeed, we consider a more general cost model in the Section IV.

In our running example, the logical query of Figure 6 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 7 and 8. The plan of Figure 7 first accesses $\texttt{confSchedule}_{(1)}$ by using a topic, and retrieves start and end dates, and city of a conference; with

```
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. 7. Logical access plan for the query of Fig. 6 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. 8. Logical access plan for the query of Fig. 6 using `hotel` first

these and the user input, all the other services can be accessed and all the comparisons can be evaluated; in particular, the remaining services may be invoked in any order, or even in parallel. The plan of Figure 8 uses a different strategy: it first invokes $\texttt{hotel}_{(2)}$ and uses its city, check-in date and check-out date to call the other services, which may, again, take place in any order or in parallel. Note that not all possible combinations of access patterns are executable with respect to the access patterns: for example, one cannot use together (in whatever order) the versions of $\texttt{confSchedule}_{(2)}$ and $\texttt{hotel}_{(1)}$ that have `City` and `To`, resp., as input fields.

B. Unanswerable queries and query expansion

For some queries it may happen that no suitable combination of services' adornments exist to obtain an executable logical access plan. In this case, it is impossible to answer the original user query, because there is at least one service whose input parameters are needed, but have not been provided by the user and cannot be taken from the output of a previously called service. Even in such a case, it may still be possible to obtain a subset of the answers to the original user query by invoking services that are not necessarily mentioned in the query, but that are available in the schema. The maximum set of answers obtainable in this way is called the *maximally contained answer*. In particular, such "off-query" services may be invoked so as to provide useful bindings for the input fields of the services in the query. This possibly lengthier process is based on information about the abstract domains of the services' attributes: an output attribute may provide a binding for an input attribute sharing the same domain. In our running example, a taxonomy of the domains in the global ontology can be used to conclude, e.g., that `confSchedule.City`, `weather.City`, `flight.From`, `flight.To`, and `hotel.City` are all locations; if these were all input fields but there was another service, say, `oldTown(City)` providing locations in output, this could be use to find some of the answers to the query.

Techniques are available to determine whether the query can be expanded with calls to off-query services to obtain a logical access plan [13], [17], or if it is definitely *unanswerable* with respect to the access patterns. In the former case, a plan retrieving the maximally contained answer may be used as a reasonable approximation of the original query. In the latter case, impossibility to answer the query is reported to the user. We note that retrieving the maximally contained answer in general requires the evaluation of a recursive query plan even if the initial query was non-recursive, but extra assumptions on the services can be made so as to always avoid recursion [18].

IV. Query optimization

A. Operation Model

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.

The derivation of the possible physical access plans for a given logical plan is driven by a number of choices suggested by an optimization strategy, namely:

- *Parallelism* among services: Depending on the restrictions imposed on the logical access plan, it might still be possible to decide whether two (or more) specific services are to be invoked in series or in parallel.
- *Chunking* of service results: Web services (especially search services) typically allow the chunking or paging of their results in output. It might thus be necessary to perform multiple *fetches* of a chunked service, in order to construct the complete result set.
- *Join strategies*: Different join strategies might be adopted to merge service results. In [19] we introduce two fundamental techniques: Nested Loop (NL) and Merge-Scan (MS) – see Figure 9. NL can be employed when there is one service that "dominates" the invocation of the other service; in this case, the exploration of the result space takes place by executing a given number of fetches of the dominated service for each output tuple coming from the dominating service. MS, instead, can be employed when there is no a priori distinction between the services to be joined; in this case, given numbers of fetches are executed in parallel for both services, and tuples in output are produced by traversing their cartesian product "diagonally". If information on how relevance decreases for the services' results is available, this might be used to determine the most convenient join strategy. For both NL and MS, the tuples in output will be returned according to a partial order combining the ranking of the two services.

Depending on these three aspects, there might be different optimization strategies. For instance, we may have a strategy that maximizes the parallelism among invoked services, or we might have a strategy that minimizes the calls to search services by choosing an appropriate join strategy.

As in [18], we call *selectivity* the average number of tuples a service produces in output in response to an input tuple.

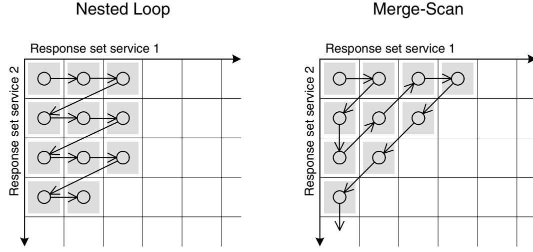

Fig. 9. The Nested Loop and Merge-Scan join strategies.

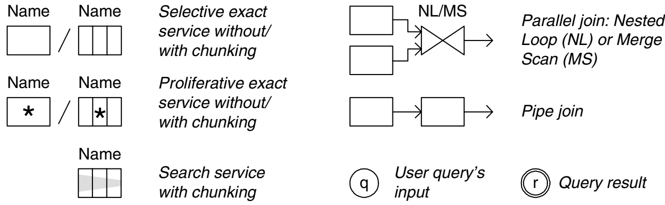

Fig. 10. Modeling notation for the graphical representation of a physical access plan.

Accordingly, a service is *selective* if its selectivity is at most 1, *proliferative* if greater than 1. Figure 10 introduces the graphical modeling notation that we use to represent a physical access plan. Selective, exact services are represented as simple boxes; proliferative, exact services 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 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. Search services are always invoked by chunking their result sets. In the physical access plan, it is important to highlight for each service whether it is invoked via chunking or not, in order to be able to estimate the cost of the service in the plan. 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).

B. Cost Model

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. One of the following cost metrics (or a linear combination thereof) may be adopted:

- a *monetary* metric, which is based on the money one needs to spend to invoke a service;

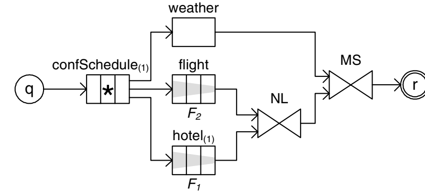

Fig. 11. Possible physical access plan for the logical access plan of Fig. 7.

- a *bottleneck* metric, which is based on the slowest (in terms of computation time) service in the query;
- a *request-response* metric, which is based on the number of service invocations; and
- a *time-to-market* metric, which is based on the time required to present the user with the first output tuple.

All cost metrics are based on the services' selectivities. Typically, exact services have a limited selectivity, while for search services it may be very large (think for instance of Google). Note that selection predicates, including join predicates, also have an intrinsic selectivity. For simplicity, we will assume that the selectivity of selection predicates is already taken into account as part of the selectivity of the node of the physical access plan they refer to.

In a given physical access plan, the use of selectivities allows us to compute for each individual service or join an average number of tuples in input and in output. This finally allows us to estimate the overall cost of the access plan in order to obtain K tuples in output of the query. The problem is typically known under the name "top-k queries" [20], [21], and several strategies have been developed to minimize the cost of query execution also in presence of ranked results [22], [23], [24], [25]. In a similar vein, also "skyline queries" [26] aim at query optimization over multiple dimensions; the most notable algorithm in this context is the Nearest Neighbors [27].

Orthogonally to the optimization strategy used to construct the physical access plan, we are developing suitable profiling techniques to derive the best *chunk size* for each of the registered, physical services. As for instance discussed in [18][2], web services typically behave differently in terms of response time or throughput depending on the adopted chunk size. Identifying the chunk size that allows the most efficient use of a service may thus speed up the overall query execution and, at the same time, also allows for the *pipelining* of the query execution. Indeed, chunking prevents the query engine from waiting for complete service responses and enables query processing over partial response sets.

C. Query Optimization Examples

If we consider the logical access plans shown in Figures 7 and 8, we can for instance derive the physical access plans of Figure 11 and 12, resp.[3] The plans make use of the

[2]We however focus on the chunking of service outputs; in [18] the focus is on chunked inputs.

[3]Note that in our graphical representation we abstract away projection and selection operations, since they are not relevant for the specification of the workflow logic underlying the physical access plan.

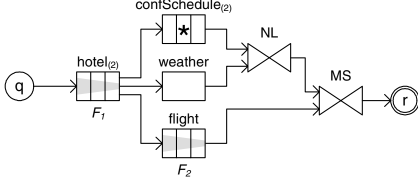

Fig. 12. Possible physical access plan for the logical access plan of Fig. 8.

selective, exact service `weather`, the proliferative, exact service `confSchedule`, and the two search services `flight` and `hotel`.

If for instance we consider the request-response (RR) cost metric to calculate the cost to obtain K answer tuples, we need to determine the expected number of invocations or fetches that need to be performed for each service. This requires knowing the selectivities of each version of exact services in the plan (i.e., `weather` and both versions of `confSchedule`). We do not consider the selectivities of search services, since, typically, we will never wholly explore their result set; therefore, the interesting parameter that characterizes such services is the number of fetches F that are to be executed for each input tuple at the service. Such number may be estimated at compile time according to the selectivities of all the services involved in the plan, their chunk sizes, and K.

Assume, for example, that for the plan of Figure 11 $\texttt{confSchedule}_{(1)}$, `weather`, `flight`, and $\texttt{hotel}_{(1)}$ require, resp., 3, 1, 3, and 4 invocations; then, the total RR cost associated to this plan is $RRcost_1 = 3 * (1 + 3 + 4) = 24$. If, instead, for the plan of Figure 12, which uses services with different adornments and in a different arrangement, $\texttt{confSchedule}_{(2)}$, `weather`, `flight`, and $\texttt{hotel}_{(2)}$ require, resp., 1, 1, 4, and 2 invocations, then the total RR cost is $RRcost_2 = 2 * (1 + 1 + 4) = 12$. Therefore, the plan of Figure 12 (P2) better optimizes the cost than that of Figure 11 (P1), according to the chosen cost metric:

$$BestPlan = \arg\min_{p \in \{P1,P2\}} RRcost(p) = P2$$

As can be seen in the above examples, *parallel* joins are computed over two output attributes (cf. the output of the services $\texttt{hotel}_{(1)}$ and `flight` in Fig. 11), whereas *pipe* joins are computed over an output and an input attribute (cf. the join between the services $\texttt{hotel}_{(2)}$ and `flight` in Fig. 12). More precisely, the pipe join requires a sequential arrangement of the two services, as the output attribute of the first service provides values in input to the second service; this necessarily implies a NL execution strategy. The parallel join, instead, can be fully parallelized, as there are no invocation dependencies among the two services to be joined.

V. 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 [28] for a recent survey).

The first studies led to develop semi-automated systems, which required a training phase performed with user intervention. Among such systems, Lixto [29] is particularly relevant. To alleviate the need for human intervention, several techniques for automatically inferring Web wrappers have been developed [30], [31], based on the observation that pages from data intensive Web sites can be grouped in classes sharing a common structure.

Pages in data-intensive sites 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* of pages in a site a collection of pages generated by the same script. We may re-formulate the problem as follows: "given a set of sample HTML pages belonging to the same class, find a hierarchical data structure (typically encoded in XML) capable to carry the same information of the original Web pages".

Arasu and Garcia-Molina proposed ExAlg [30], 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. The basic idea of ExAlg is to deduce the unknown template by using the concepts of equivalence classes and differentiating roles of tokens. It works in two stages; in the first stage it discovers sets of tokens associated with the same type constructor in the template used to create the input pages; in the second stage it uses the token sets to deduce the template and extract the values encoded in the pages. Roles of tokens are differentiated using the context in which they occur in the page.

RoadRunner [31] 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 fact that regular expressions cannot be learned from positive examples alone, and the high complexity of the learning even in the presence of additional information, limits the applicability of the traditional grammar inference techniques to Web sites, and motivates resorting to a number of pragmatical approaches. Still, RoadRunner avoids relying on user-specified examples, requires no interaction with the user during the wrapper generation process, and requires no a priori knowledge about the page contents, i.e., it doesn't need to know the schema according to which data are organized in the HTML pages; such schema will be inferred along with the wrapper

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 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.

VI. CONCLUSION

In this paper, we have proposed a framework for answering multi-domain queries over Web-accessible data sources, including wrapped Web sites and heterogeneous search services, which return their results in ranked order. Users queries, expressed over a conceptual abstract view, are rewritten into query plans, i.e., sequences of service invocations, which are optimized taking into account several characterizations of the involved services. Among the distinctive aspects of our approach, we mention query optimization based on several possible cost metrics and the capability of joining ranked result lists.

The end-user interface most suitable to this approach is a simple form on the free arguments of the query. In our future work, we envision offering to users a lower-level query interface similar to Yahoo pipes, enabling the direct composition of services based upon their adornment. Experiments will then indicate to us whether such a tool is sufficiently user-friendly. We will also experiment the use of higher-level interfaces, by using ontological knowledge for query formulation. At the current stage of the project we have developed several of the components of the NGS architecture, including software components performing query mapping, expansion and optimization, and a collection of wrapping generation tools. In the next stages of the project we will integrate the software components into a complete demonstrator of the NGS approach.

ACKNOWLEDGEMENTS

All authors acknowledge support from Italian PRIN project "New technologies and tools for the integration of Web search services".

REFERENCES

[1] A. Y. Halevy, "Answering queries using views: A survey," *VLDB Journal*, vol. 10, no. 4, pp. 270–294, 2001.
[2] J. D. Ullman, "Information integration using logical views," in *Proc. of ICDT'97*, ser. LNCS, vol. 1186. Springer, 1997, pp. 19–40.
[3] M. Lenzerini, "Data integration: A theoretical perspective." in *Proc. of PODS 2002*, 2002, pp. 233–246.
[4] S. Abiteboul and O. Duschka, "Complexity of answering queries using materialized views," in *Proc. of PODS'98*, 1998, pp. 254–265.
[5] G. Grahne and A. O. Mendelzon, "Tableau techniques for querying information sources through global schemas," in *Proc. of ICDT'99*, ser. LNCS, vol. 1540. Springer, 1999, pp. 332–347.
[6] R. van der Meyden, "Logical approaches to incomplete information," in *Logics for Databases and Information Systems*, J. Chomicki and G. Saake, Eds. Kluwer Academic Publishers, 1998, pp. 307–356.
[7] A. Calì, D. Calvanese, G. De Giacomo, and M. Lenzerini, "Data integration under integrity constraints," *Information Systems*, vol. 29, pp. 147–163, 2004.
[8] ——, "On the expressive power of data integration systems," in *Proc. of ER 2002*, 2002.
[9] A. Calì, D. Lembo, and R. Rosati, "Query rewriting and answering under constraints in data integration systems," in *Proc. of IJCAI 2003*, 2003, pp. 16–21.
[10] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati, "DL-Lite: Tractable description logics for ontologies," in *Proc. of AAAI 2005*, 2005, pp. 602–607.
[11] ——, "Tailoring OWL for data intensive ontologies," in *Proc. of the Workshop on OWL: Experiences and Directions (OWLED 2005)*, 2005.
[12] ——, "Data complexity of query answering in description logics," in *Proc. of KR 2006*, 2006, pp. 260–270.
[13] A. Rajaraman, Y. Sagiv, and J. D. Ullman, "Answering queries using templates with binding patterns," in *Proc. of PODS'95*, 1995.
[14] O. M. Duschka and A. Y. Levy, "Recursive plans for information gathering," in *Proc. of IJCAI'97*, 1997, pp. 778–784.
[15] A. Nash and B. Ludäscher, "Processing first-order queries under limited access patterns," in *Proc. of PODS 2004*, 2004, pp. 307–318.
[16] G. Yang, M. Kifer, and V. K. Chaudhri, "Efficiently ordering subgoals with access constraints," in *Proc. of PODS 2006*, 2006, pp. 22–22.
[17] A. Calì and D. Martinenghi, "Querying data under access limitations," in *Proc. of ICDE 2008*, to appear.
[18] U. Srivastava, K. Munagala, J. Widom, and R. Motwani, "Query optimization over web services," in *VLDB-06*, 2006, pp. 355–366.
[19] D. Braga, A. Campi, S. Ceri, and A. Raffio, "Joining the results of heterogeneous search engines," *Information Systems*, submitted to.
[20] R. Fagin, "Combining fuzzy information from multiple systems," in *Proc. of PODS 96*, 1996, pp. 216–226.
[21] M. J. Carey and D. Kossmann, "On saying "Enough already!" in SQL," 1997, pp. 219–230. [Online]. Available: citeseer.ist.psu.edu/carey97saying.html
[22] N. Bruno, S. Chaudhuri, and L. Gravano, "Top-k selection queries over relational databases: Mapping strategies and performance evaluation." *ACM Trans. Database Syst.*, vol. 27, no. 2, pp. 153–187, 2002.
[23] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter, "Supporting incremental join queries on ranked inputs," in *Proc. of VLDB 2001*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 281–290.
[24] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid, "Supporting top-k join queries in relational databases." *VLDB J.*, vol. 13, no. 3, pp. 207–221, 2004.
[25] N. Mamoulis, M. L. Yiu, K. H. Cheng, and D. W. Cheung, "Efficient top-k aggregation of ranked inputs," *ACM Trans. Database Syst.*, vol. 32, no. 3, p. 19, 2007.
[26] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator," in *Proc. of ICDE 2001*, 2001, pp. 421–430.
[27] D. Kossmann, F. Ramsak, and S. Rost, "Shooting stars in the sky: an online algorithm for skyline queries," in *Proc. of VLDB 2002*, 2002, pp. 275–286.
[28] 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.
[29] R. Baumgartner, S. Flesca, and G. Gottlob, "Supervised wrapper generation with lixto," in *The VLDB Journal*, 2001, pp. 715–716. [Online]. Available: citeseer.ist.psu.edu/baumgartner01supervised.html
[30] A. Arasu, H. Garcia-Molina, and S. University, "Extracting structured data from web pages," in *Proc. of ACM SIGMOD*, 2003, pp. 337–348.
[31] 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.
[32] V. Crescenzi and G. Mecca, "Automatic information extraction from large websites," *J. ACM*, vol. 51, no. 5, pp. 731–779, 2004.