Joining the Results of Heterogeneous Search Engines

Daniele Braga    Alessandro Campi    Stefano Ceri    Alessandro Raffio
Politecnico di Milano - Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci 32, 20133 Milano, Italy
braga | campi | ceri | raffio @elet.polimi.it

Abstract
In recent years, while search engines have become more and more powerful, several specialised search engines have been developed for different domains (e.g., geographic services, services dedicated to specific business sectors, library services, and so on). While such services beat generic search engines in their specific domain, they do not enable cross-references; therefore, they are of little use when queries require input from two or more of such services (e.g., “vegetarian restaurants in the surroundings of San Francisco”). In this paper, we study how to join heterogeneous search engines and get a unique answer that satisfies conjunctive queries, where each query can be routed to a specialised engine. The paper includes both the theoretical framework for stating such problem and the description of pragmatic solutions based on Web Services technology.

1 Introduction
The current evolution of the Web is characterized by an increasing number of search engines, ranging from generic ones (such as Google) to domain-specific ones (such as geo-localization services, or on-line recommendation systems for specific goods, or library search systems). While each search engine can be separately used to issue focused queries, their intrinsic limit is the inability to go beyond the purpose for which they have been developed; however, users are often interested into complex queries, ranging over multiple domains. Such queries can be only answered, at the current state of art, by a deep involvement of a knowledgeable user, who inspects search engines one at a time, feeding the results of one search as input to the next one. However, in an ideal scenario, users do not want to be bothered by distinctions between many searching systems and desire to have one common interface available for querying them; moreover, while they can accept a complex interaction when their query is rather complex, they certainly do not want to “cut-and-paste” query results into query inputs, as such approach is time-consuming and yields to imprecise results.

The focus of this paper is to develop techniques for integrating the results extracted from several existing search engines. Our main idea is to design a system offering a common interface to several, known search services such that a user query can be decomposed and its components can be routed to many of them in parallel; the system is then capable of performing the integration of results extracted by each of them. Given that each result is independently built, this approach consists in joining results into a composite result, to be presented to the user. The recent availability of Web Services and XML as a means for generalised interoperability makes such an objective quite feasible.

In essence, this paper opens up the new problem of “joining” search engine results. Such a join is indeed far more complex than conventional joins over tables in relational databases: search engine results are ranked lists of complex XML structures, and the pairwise join of the elements of such lists is far from trivial. However, “join” is still the underlying paradigm, as we can talk about “join methods” which define the way in which ranked lists are examined in order to produce the “join results”.

1.1 Relevant Examples
Our ideal query involves several investigation dimensions, each covered by a given search service. Examples of nontrivial queries that address orthogonal dimensions are the following:
- Find a good vegetarian restaurant at approximately 30 miles from San Francisco.
- Find all VLDB authors from ETH Zurich.
Any human actor recognizes at a glance that these queries can be considered as conjunctions of simpler queries over independent dimensions. The first one requires combining a geographic Web Service (say, MapPoint) and a recommending service expert in restaurants (say the Michelin Guide, or perhaps a more specific service restricting to Vegetarian Restaurants), the second one requires combining a Web Service dealing about publications (say, DBLP) and the staff of a given Faculty, as provided by a given wrapping service over a Web site.

Another well-known source of “challenging” queries currently not supported by search engines is from Weikum’s [25]:

Q1: Which professors from Saarbruecken in Germany teach information retrieval and do research on XML?
Q2: Which gene expression data from Barrett tissue in the esophagus exhibit high levels of gene A01g? And are there any metabolic models for acid reflux that could be related to the expression data?
Q3: What are the most important research results on large deviation theory?
Q4: Which drama has a scene in which a woman makes a prophecy to a Scottish nobleman that he will become king?
Q5: Who was the French woman that I met in a program committee meeting where Paolo Atzeni was the PC chair?

We declare queries 2, 3, and 4 as unfeasible by our method, because they require a deep understanding of natural language and of the domain. Instead, queries 1 and 5 might be addressed, with suitable sources, and they look similar to our running example.

1.2 Technological Scenario

The main technological software revolution of the last years is the development of Web Services; this technology allows applications to exchange information effectively and represents the turning point of several previous approaches to systems interoperability. Several Web Services are already available on the Web, not only for specific querying purposes, such as searching book catalogues or dynamically generating maps from geographic information systems, but also for querying general purpose search engines (such as Google) from within other applications. In addition, it is becoming possible to query those information sources which do not expose a Web Service interface by means of suitable wrappers, which monitor the data contained in such sources and provide a Web Service-based interface for querying the wrapped data [4, 9]. This opportunity broadens the scope of the approach described in this paper and opens interesting scenarios for empowering Web searches.

This paper discusses the issue of integrating search services so as to answer complex queries by leveraging those (simpler) search services, which are already available. More long-term research concerns Web Service Composition (i.e., the ability to combine Web Services which are capable of performing tasks so as to build complex workflows) and Semantic Web Services [2, 1] (i.e., the ability to select the services that best match a user’s goal where both the goal and the services are semantically described by means of knowledge bases or ontologies). Compared to the above long-term goals, the objective of effectively joining results from different join engines is much easier.

1.3 Approach

Let us consider again the problem of finding vegetarian restaurants within 30 miles from San Francisco. This would require to query a geo-localization service for all the locations within 30 miles from San Francisco and to query a dedicated service (or a generic one, if not available) for a list of vegetarian restaurants. Then, results of the two services must be “joined”, extracting vegetarian restaurants at the selected locations; it is assumed that information about restaurants includes their location, although perhaps the two encodings of locations in the two services are different.

In this paper, we will not focus on how the initial user query can be decomposed into the two subqueries. In the simplest case, a knowledgeable user could select the appropriate services out of a given set of services known to our system, and then interact with each of them by submitting the subqueries. With a more advanced and user-friendly interface, subquery decomposition could be done automatically from a single, initial query, by matching query terms to service descriptions, possibly helped by ontologies; this is the subject of parallel ongoing research.

This paper instead focuses on the join of results produced by two services, and considers such operation as the building block for enabling the use of several services in sequence. The join takes place among ranked lists of XML fragments, where each fragment represents one individual resource descriptor within the search result. Compatibility between two fragments is determined by matching given XML elements; these are known a-priori, as the XML schemas of Web Service responses are known; however, the matching of two fragments may be “stronger” or “weaker”, as it involves value and string comparison (e.g., identity, compatibility, and so on).

Results returned to users will therefore be based upon the original ranking of fragments and upon the result of matching; presenting the result requires composing the two entries according to a composition strategy which is known for every pair of services. Of course, results are presented to users in “batches”, as with conventional search engines, and most batches will not be computed unless the user interacts with the system asking for “more” results. Normally, the join is arrested when relevance is below a given threshold or
when enough elements are produced in output; if the
user is not satisfied, then the search is continued with
the following elements in the relevance ranking. Thus,
the stronger analogy is with “top-k” joins of relational
databases [8, 15].

In addition, we require the system to present re-
results to the user in an order that is sufficiently close
to their “result ranking”, i.e. the ranking that is com-
pined by the system for each result entry. We will see
that outputting results in exact ranking order is not
convenient, but such ordering can be approximated.

When presenting a new kind of join, it is essential
to define metrics for evaluating its cost. In our setting,
costs depend on two factors: the interaction with Web
Services through the execution of request-response
pairs, and the execution of several XML fragments
comparisons after each new Web Service call. These
recall the well-known join cost factors of join optimiz-
er: i/o costs (now replaced by request/response pairs)
and cpu costs (now replaced by the computational
costs needed for joining XML fragments, when the
matching requires textual comparison and/or domain-
specific knowledge). In principle, the two costs adds
up; however, we may easily foresee scenarios where the
former or the latter cost factor dominates, and we will
consider these extreme cases as most representative.

1.4 Paper organization

Section 2 presents the problem statement, Section 3
is dedicated to join methods, and Section 4 presents
examples and experimental results. The paper is con-
cluded with related works and future works.

2 Problem statement

A query \( q \subseteq K \) is a set of keywords taken from an
alphabet \( K \). Given \( n \) services \( S_i, i = 1..n \) used for solving
\( q \), we assume that \( q \) is decomposed into \( n \) subsets of
keywords, possibly overlapping, such that
\( q = \bigcup_{i=1}^{n} q_i \),
and that each query \( q_i \) is evaluated by the correspond-
ing service \( S_i \). The decomposition of \( q \) into subqueries
\( q_i \), is beyond the scope of our research. We next for-
mally define service responses and a binary join be-
tween two such responses.

2.1 Definition of Service Responses

Given a generic service \( S \) and a query \( q \), we denote
as \( X \) the response returned by \( S \) on \( q \); we say that \( q \)
produces \( X \) on \( S \), or, more compactly, \( S \models_q X \). Since
all the services \( S \) dealt with in the paper expose a Web
Service interface, we also assume that \( X \) is a sequence
of valid XML fragments w.r.t. the schema definitions
contained in the WSDL description of the request-
response primitive for \( S \). We denote as \( N_X \) the car-
dinality of \( X \), i.e. the number of items in the result.

Each item \( x_i \in X \) is provided with a scoring value
\( \rho_i \in [0, 1] \), which is an estimate of the relevance of \( x_i \)
w.r.t. the query \( q \). We assume that services return
items \( x_i \) in scoring order.

Given that each item \( x_i \) is a fragment, we can
also think of it as a collection of PCDATA and at-
tribute values, where each such value is defined by
a suitable path expression, valid w.r.t. the frag-
ment’s schema. Formally \(^1\), \( x_i = (\langle \text{val}_{ij}, \rho_i \rangle, \) with
\( \text{val}_{ij} = (\text{path}_j, \langle \text{v}_{ijk} \rangle) \), where \( \langle \text{v}_{ijk} \rangle \) is the list of val-
ues obtained by the evaluation of the path expression
\( \text{path}_j \) within the XML fragment \( x_i \).

2.2 Join of two responses

Without loss of generality, we can refer to the case
of join responses from two services. Given two serv-
eses \( S_X \) and \( S_Y \) and a query \( q = q_X \cup q_Y \), such that
\( S_X \models_{q_X} X \) and \( S_Y \models_{q_Y} Y \), we need to join \( X \) and \( Y \).
Borrowing the notation from classical Relational Al-
gebra, we write \( R = X \bowtie Y \); where, as usual, \( \bowtie \) is the
join operator. Indeed, the join of \( X \) and \( Y \) is not as
simple as a join within relational databases, since we
are dealing with structured elements and not with flat
tuples.

The result of the join operation is a sequence \( R \),
which contains the elements \( r_k = \langle c(x_i, y_j), \rho_k \rangle \)
which answer the original query \( q \). The term \( \rho_k \) rep-resents an estimate of the relevance of \( r_k \) w.r.t. \( q \); \( x_i \) and \( y_j \)
are selected items taken from \( X \) and \( Y \) respectively;
\( c \) denotes a composition operation to be properly ap-
plied to the components of \( x_i \) and \( y_j \), which is set up
in the service registration phase.

We can think of many different ways to register a
service within our framework, that lead to different
techniques for calculating \( r_k \) and \( \rho_k \) out of each cou-
ples \( (x_i, y_j) \) - i.e. ways to define function \( c(\cdot) \), ranging
from naive direct coupling performed by human inter-
vention, feasible when the framework deals with few
predefined services, to sophisticated semantic-aware
techniques, which are useful to build a flexible sys-
tem capable of addressing many different services, dy-
namically identified and coupled. The registration of
services goes beyond the scope of our research; we sim-
ply assume the proper coupling schema between \( path_{ih} \)
and \( path_{jk} \) as available in the environment, contain-
ing a weight \( c_{hh} \) and the indication of the matching
operator to be applied for the specific couple of paths.

In turn, the matching between each single pair of
XML values (elements or attributes), as identified by
a couple of paths, can take place using several different
operators; such operators can be organised in a hier-
archy, rooted on (typed) equality, that includes text
containment, partial textual matching, and possibly
ontology-driven term similarity. Clearly, when com-

\(^1\)Throughout the paper curly braces like \( \{e_i\} \) denote a set of
elements \( e_i \), angle brackets like \( \langle e_i \rangle \) denote an ordered sequence
of homogeneous elements, regular brackets like \( [e_1, ..., e_n] \) denote
a \( n \)-uple of heterogeneous elements, and square brackets like
\([e_1, e_2]\) denote an interval between two values.
paring two strings, (e.g., the “location” of a Restaurant), their identity yields to a higher matching index than partial string matching or term similarity of the same XML values. It should be possible to evaluate each pair with a different operator, as well as to use the same one in the entire process. Of course the computational cost of different coupling schemas and different operators is substantially different, as it is much faster to test shallow identity on few components than to apply complex metrics to all possible combinations of values, taking into account semantic proximity and possibly multilinguism.

2.3 Matching strategy

The estimation of $\rho_k$ requires comparing the two elements $x_i$ and $y_j$ according to a given matching strategy. The result of the match operation is a match index $m_{ij} \in [1,0]$, expressing the probability that the two elements $x_i$ and $y_j$ describe the same resource or real world fact or event. Note that match indexes have no counterpart in the relational join, as two values being compared are either identical ($m_{ij} = 1$) or different ($m_{ij} = 0$). Then, the relevance $\rho_k$ is simply given by the product\(^2\) of the two scoring orders of the fragments $x_i$ and $y_j$, produced by the search engines, and the match index, produced by the matcher:

$$\rho_k = \rho_i \times \rho_j \times m_{ij}$$

Note that every term in the right side of the above formula ranges in the $[0,1]$ interval and therefore also $\rho_k \in [0,1]$.

2.4 Efficiency measures

Given the above definitions, it is possible to define an abstract notion of optimality: a join strategy is optimal, relative to a given constant value $\overline{\rho}$, if it produces in output elements $r_k = (c(x_i, y_j), \rho_k)$ that answer the original query $q$ such that $\rho_k \geq \overline{\rho}$, in exact ranking order, with the minimum cost, either in terms of number of request-responses addressed to the two search services or in terms of time spent in computing the degree of similarity. Extraction-optimality guarantees that a result entry whose ranking falls below a given threshold will not be presented, but then presents results above threshold in the order in which they are extracted from the services; therefore, the dataflow of results produced to the user is not blocking (abusing of the terminology which is typical of query streams), and results can be batched to users while they are extracted from the search engines.

2.5 Scenarios

Concerning the cost model, we consider the following scenarios:

A. Cost of request-response execution dominated by join execution. We assume that joining the blocks that become progressively available is the expensive operation, e.g. because each element comparison requires itself a call to a semantic Web Service.

B. Cost of join execution dominated by request-response execution. We assume that once a block of XML elements is retrieved as the effect of a request-response to services, then join requires simple main-memory comparison operations and can be neglected.

---

\(^2\)We assume for simplicity that the combination of such values is done by multiplication. Of course any other triangular norm can be applied, so as to affect the way in which the combined weight decreases w.r.t. the decrease of each single weight.
Concerning the availability of rankings, we start by assuming that ranking be available for both services as one of the parameters which are returned together with each XML entry. However, ranking can be unavailable on either one or both services. In this case, we further characterize the service according to its expected behavior in the following two classes:

1. Step ranking. We assume that, by performing a limited number K of request-response, most of the relevant entries can be retrieved, because the entry relevance decreases with a step, and such deep step is normally captured by doing k request-responses. For simplicity we assume K to be a parameter associated with the service.

2. Linear ranking. We assume that the entry relevance decreases roughly linearly, with no step.

Note that the unavailability of the ranking function to the planner does not affect our basic assumption that the search services return their result in ranking order, but simply considers the fact that this function may be opaque to the planner.

Of course, cases A/B and 1/2 represent extreme behaviors; they are defined so as to enable us discussing particular techniques which would be convenient if they occur, and then performing experiments with actual services so as to check experimentally which technique better responds to the specific Web Services. In the next section we then define cases A and B, and then the variations A1, A2, B1, B2 when rankings are unavailable under the two cost assumptions.

3 Join strategies

As mentioned above, we assume that Web Services return results in decreasing ranking order. Such ranking can be observed while the elements are being retrieved one “batch” (o block) at a time, by reading their ranking values, and is possibly measured in terms of an accessible figure.

We can represent the blocks of the results extracted from two services $S_X$ and $S_Y$ over the axes of a Cartesian plan, such that on each axis the ranking order of the blocks decreases from the origin down to the end of the list (see Figure 1). Each point $P$ in the plan represents a couple $(x, y)$ which is a candidate $r_k \in R = X \bowtie Y$. The plan is divided into rectangles of size $n_X \times n_Y$, where $n_i$ represents the number of results contained in each block returned by $S_i$. We call tile $t_{ij}$ the rectangular region that contains the points relative to blocks $b_X$ and $b_Y$. Two tiles are said to be adjacent if they have one edge in common.

The plan is a model of the search space to be explored by the joiner. Each rectangular region of size $m \times n$ represents the part of the search space that can be inspected after performing $m$ request-responses to $S_X$ and $n$ request-responses to $S_Y$. Therefore, achieving extraction-optimality requires a suitable exploration strategy for such search space, which guides a “careful scan” of the result lists.

As aforementioned, two main scenarios are possible, depending on which is the dominating cost in the scan: service access or join execution. The rest of this section discusses how to manage these scenarios.

3.1 Dominance of matching costs

When matching cost dominates (case A according to 2.5), the best strategy is to choose at each step the tile with highest ranking, as this has the highest potential of producing results.

3.1.1 Tile Extraction Optimal (TEO) method

As we are interested in exploring the most promising tiles first, we define an estimate $w_{ij}$ of the expected relevance for the results of the join operation performed on the points of tile $t_{ij}$. For simplicity, we use the first point of each tile (i.e., the one with smaller coordinates) as representative for the tile. We also use the ranking $p$ of the last element returned by each service call as an estimate of the ranking of the first element that would be returned by the next call to that service.

All exploration algorithms begin by initializing two sets, $\mathcal{E} = \emptyset$ and $\mathcal{U}$ initially containing all possible tiles, representing the explored and unexplored tiles respectively. During the exploration, tiles progressively migrate from $\mathcal{U}$ to $\mathcal{E}$, until the end of the search.

We also define a set $\mathcal{C} \subseteq \mathcal{U}$ of candidates: tiles in $\mathcal{C}$ are all tiles $t_{ij}$ which are adjacent to some tile in $\mathcal{E}$. Further, a tile in $\mathcal{C}$ is dominated by another tile of the same set when it is adjacent to another tile in $\mathcal{C}$ with smaller index sum. Due to the assumed monotonicity of the relevance estimates of the service responses along each axis, dominated tiles have a lower expected relevance than their respective dominating tiles. Figure 2 shows four possible configurations of the Cartesian plan, where tiles in $\mathcal{E}$ are represented in

![Figure 2: Deciding the next step in the scan](image-url)
black, tiles in $\mathcal{U}$ are represented in white, and tiles in $\mathcal{C}$ which are not dominated are marked with a dot.

Referring to Figure 2, we can equivalently say that (a) each column or row contains at most one dotted tile and (b) for each dotted tile all tiles whose indexes are (both) lower or equal belong to $\mathcal{E}$ and all tiles whose indexes are (both) higher or equal belong to $\mathcal{U}$. Therefore, after $n+m$ request-response operations, it is guaranteed that the cardinality of the set of nondominated candidates is between 2 and $min(m,n)+1$.

Given the notion of dominance between tiles as defined above, the algorithm Tile Extraction Optimal (TEO) for scanning the search space is as follows:

1. Perform an initial request-response to each of $S_X$ and $S_Y$, thus retrieving two blocks of results, namely $(b_{X1}, b_{Y1})$; they correspond to the first tile in the search space.
2. Compute $\mathcal{E}$, $\mathcal{U}$, and $\mathcal{C}$; Compute the estimate of rankings $\rho$ for all nondominated tiles in $\mathcal{C}$.
3. Choose in $\mathcal{C}$ the tile $T$ with the highest estimate for the average expected relevance $a_{ij}$.
4. If $T$ has either one of its indexes $i$, $j$ set to 1, then perform a request-response to the relevant service.
5. Perform the join operation between the results of the web service calls relative to the chosen tile, and output the results that are above the relevance threshold.
6. Goto step 2 unless (a) the search space is exhausted or (b) the user stops the search.

The above algorithm is tile-extraction-optimal because it extracts tiles in decreasing order of the tile’s estimated rankings\(^3\). Strictly speaking, extraction-optimality as defined in Section 2.5 would require operating element by element, by “anticipating” request-responses, i.e. performing two initial request-responses at step (1), and then building the joins in all nondominated elements of $\mathcal{C}$, outputting results in decreasing ranking of $\rho_i \times \rho_j$. However, the proposed algorithm is indeed much simpler and is extraction-optimal under the two simplifying assumptions of selectivity estimates.

### 3.2 Dominance of service requests

In this scenario (case B in 2.5), we assume that request-response execution dominates over matching costs; therefore, the most efficient approach is to compare all the elements in all the retrieved tiles in all the possible combinations before submitting another service call.

\(^3\)We recall that we estimate the ranking of all the elements of a tile as its first element and that we use, for candidate tiles requesting a new Web Service call, the ranking of the last element retrieved by the last Web Service call.

#### 3.2.1 R-Shape method

As the cost of joining all the couples in a tile is dominated by the cost of performing one single i/o operation, the choice reduces to choosing at each request-response which service is to be invoked next. After retrieving each batch of new results, the join is performed between the new items and all the cached items, thus exploring an entire row (or column) of tiles. Note that after $K$ request-response operations performed over service $S_X$, one call to $S_Y$ allows the joiner to explore $K$ new tiles. Figure 3A shows by means of numbering the order in which the first regions of the search space are explored, in a case in which the ranking trends suggest that the most convenient schedule for the first service calls is the following: $S_X$, $S_Y$, $S_X$, $S_Y$, $S_X$, $S_Y$, $S_Y$, $S_X$.

It should be noted that a strong asymmetry in the ranking of the two services may lead to a “long and thin” rectangular area composed of the already explored tiles. This degenerates, in the worst case, to addressing all the calls to one service only (except for the first two calls, which are always alternated so as to have at least one tile for starting the exploration). This particular case, shown in Figure 3B has the disadvantage that each i/o only adds one tile; to avoid this, one may want to (heuristically) call a few batches on both services before applying the algorithm. Some further heuristic opportunities for trying to anticipate the retrieval of tiles which are likely to produce good matches will be illustrated in Section 4.4.

### 3.3 Join Strategies in the Lack of Rankings

While in the previous sections we assumed that search services would return explicit ranking values, in many real-life situations services return no ranking; however, they do return results according to an internally-computed ranking. In these cases, neither TEO nor R-shape are applicable, and therefore one has to resort to simpler join strategies. We consider two extreme cases, corresponding to possible ranking trends:

- In the former case, we assume that at least one of the two trends includes a step separating matching values from nonmatching values.

Figure 3: R-Shape

![R-Shape](image-url)
• In the latter case, we assume that both trends decrease linearly.

The former case is served by the same method, consisting in anticipating the execution of the Web Service calls up to the determination of the step (that can normally be guessed by a test associated with the Web Service itself). The latter case has instead to be approached in two different ways depending on the dominant cost; we will therefore first consider the matching costs as dominant and then the service call costs as dominant.

3.3.1 Nested Loop Method

The nested loop method is suitable when the results of one search engine exhibits a clear “step”; in such case, we assume that the ranking suddenly drops from a high value (approximated as 1) to a very low value (approximated as 0). The corresponding best exploration strategy of the search space reminds of the “nested-loop” method for relational joins; the exploration consists of extracting all the blocks corresponding to the high ranking values of the “step” engine, and then extracting the blocks of the other service in ranking order, thereby producing join results. This method is represented in Figure 4(a).

3.3.2 Merge-Scan Method

The merge-scan method is indicated in the absence of information about a clear “step” in the ranking of results. Then, one should assume that rankings decrease linearly. The corresponding best exploration strategy of the search space reminds of the “merge-scan” method for relational joins; the exploration consists in moving “diagonally” in the Cartesian plan, as shown in Figure 4(b), where the arrows indicate the order in which the tiles are chosen starting from $t_{11}$.

3.3.3 S-Shape Method

The square shape method translates the merge-scan case to the situation where matching costs are negligible; therefore, as in the previous case, Web Services are called in alternation; however, the entire set of tiles (now building a square rather than rectangular shape) are considered. Figure 4(c) shows this join strategy.

4 Experimental Evaluation

The research described in this paper is part of a project dedicated to the development of a search engine integration framework. We initially present the overall architecture and then the prototype that we have implemented so far, with which we have conducted the experiments described in Section 4.2.

4.1 Joiner Architecture

Figure 5 shows the main components of the search engine integration framework:

- The **user interface** takes in input queries from the user in the format of collections of terms and shows the results as output; with an interactive adaptation, it allows users to indicate their navigation preferences through simple interactions.

- The **query decomposer** reduces the original query into several (possibly overlapping) subqueries and identifies the services that can better address each subquery, using the information about registered search services coming from the search service directory.

- The **query decomposer** reduces the original query into several (possibly overlapping) subqueries and identifies the services that can better address each subquery, using the information about registered search services coming from the search service directory.

- The **matcher** performs the join of the elements in a given tile, producing the result entries; the choice of the matching algorithm depends on the correlation of each pair of registered services, as provided by the directory service.
Our prototype takes in input two queries, a specification of the composition function $c(\cdot)$ suitable for matching two specific services, and additional information about the two services (such as WSDL descriptions and other meta-data); with such input, the tool performs service calls, joins the partial results according to one of the strategies described in Section 3, and outputs the combined results in a stream, as soon as they are ready (in extraction order). The initial query is decomposed manually into subqueries; all aspects concerning the user interface and the Web Services registration will be addressed by future work.

4.2 Experiments

We conducted a series of experiments, using some of the most popular search services available on the Web. Our experiments compare the various join methods by plotting the number of service calls and the number of explored tiles as functions of the number of retrieved "good" results (that are defined as such by knowledgeable users). This kind of visualization allows one to rapidly reckon how the "cost" of interesting results grows with the number of results, as well as to easily compare the "extraction optimality" of different strategies. We considered the methods discussed in the previous section within the context of both cost models described in section 2.5.

Unfortunately, conventional search services do not provide explicit ranking values with their results; thus, in our experiments we have generated such rankings manually by inspecting the XML fragments (e.g., ranking "recent" publications by their years, or points in the space by their distance from given locations) so as field-test TEO or R-shape together with the other approaches.

4.2.1 Typical restaurants close to a given city

Our first use case is concerned with the search of restaurants with a peculiar characteristic X in the surroundings of a location Y. This task requires to in-
egrate the results collected from a Geographic Information System for locating cities around Y (let’s say within 20 miles), and another service for a list of possible typical restaurants (which exhibit the X characteristics).

For this purpose, we use the GeoPlace service to locate towns and Amazon service to get a list of typical restaurants. The matching is based upon string equality of a single element (the zip code attribute of elements returned by GeoPlace must be the same as the “postalCode” attribute returned by Amazon). Thus, the service composition function c(·) is quite simple: only two path expressions are involved (the element tagged toPlace and the attribute named postalCode) and the matching method is strict string equality. Figure 6 shows two (matching) fragments of the responses that are retrieved from the two services (according to their WSDL description); this very simple test, however, enables us to allocate a restaurant within a given geographic area.

The Amazon service returns a list of restaurants; ranking values for this list can be approximated by a constant function, since the service returns only the restaurants that match the given query (they either have the X characteristic or they do not). GeoPlace, instead, returns a short list of towns within the 20 mile range from Y, ordered by distance. In this case, the rank can be easily computed by means of a distance function, giving score 1 to central places and 0 to places 20 miles away.

This example corresponds to the B scenario (as classified in Section 2.5), where the cost is dominated by the interaction with the remote services: indeed, the join operation is trivial. The ranking trend of the Amazon service presents a step, while the GeoPlace service’s rankings gradually decrease.

The Amazon service returns a list of restaurants; ranking values for this list can be approximated by a constant function, since the service returns only the restaurants that match the given query (they either have the X characteristic or they do not). GeoPlace, instead, returns a short list of towns within the 20 mile range from Y, ordered by distance. In this case, the rank can be easily computed by means of a distance function, giving score 1 to central places and 0 to places 20 miles away.

This example corresponds to the B scenario (as classified in Section 2.5), where the cost is dominated by the interaction with the remote services: indeed, the join operation is trivial. The ranking trend of the Amazon service presents a step, while the GeoPlace service’s rankings gradually decrease.

Figure 7: Pizza near San Francisco

Given these ranking trends, we expect the nested loop method to be the most appropriate. The method matches each city, starting from the closer ones, with all the restaurants; in this way matching restaurants are progressively extracted in a proximity decreasing order. Figure 7 confirms our intuition, as the nested loop method gets all the 6 matching restaurants by means of fewer visits. The method is compared to the merge-scan and horizontal nested-loop navigation of the same result space, shown next to the diagram. Note that all methods converge to the same final result of retrieving 6 restaurants.

Figures 8 and 9 show the same kind of query with different parameters for the city and type of restaurant. Note that merge-scan behaves rather well with example 8 - as it finds more restaurants relative to the first 6 request-responses - but in general the nested-loop method appears to dominate the three examples.

4.2.2 Retrieving books

The second example deals with the search of books authored by researchers who are “specialised” in the book’s subject, i.e., who have also published papers on that topic. This task requires to integrate the results collected from the dblp online citation archive (in order to retrieve a list of papers) and a Web Service made available by amazon.com giving information concerning the book’s subjects. The matching operations for this query are quite expensive: the service composition function needs to (1) check the equality of the authors and (2) check the semantic similarity between the title of a book and the title of a paper. While the former operation is quite inexpensive, the latter is performed by means of calls to a Web Service which

---

4The DBLP online service only allows retrieving the whole archive as a single XML document; we have wrapped the service so as to provide basic query capabilities, according to what anticipated in section 1.2.
provides semantic mappings between terms; for this operation we relied on *MoMiS* [5], a semantic schema mapper developed by the University of Modena.

Due to the fact that many calls to an external service occur within the join operation, the cost model for this example is that of scenario A: the cost of i/o is dominated by join execution. Since the ranking functions of the results are opaque, the best approach is to perform a merge-scan navigation, as shown in figure 10.

![Figure 10: Books vs request-responses](image)

### 4.3 An anomaly

The theory discussed so far demonstrates to be unable to capture the example described next.

We queried *amazon.com* and *dblp* for retrieving books on a given topic (say, databases), recently written by authors who also recently published in a particular conference (say, VLDB). Both Amazon and dblp can return results ordered by year of publication, thus we can compute the ranking value for each item as the publication year divided by the current year. Therefore, the navigation within the search space can be driven by the expected ranking values, by means of the Tile Extraction Optimal algorithm.

![Figure 11: Recent books about databases by active VLDB authors](image)

The table in the right part of figure 11 shows the first portion of the results space (the one nearest to the origin of the axes). Inside this area, 15 books matched our criteria, most of which are clustered in a specific region. This example is peculiar because of the presence of a particular author who wrote many books on a particular subject. The “anomaly” is the presence of an element (an author) in one of the results of the services which is per se hardly distinguishable from other “near” authors but, differently from them, causes several hits with a high global relevance, as he is the author of many books of interest. This outcome (the high relevance of some couples) could not be inferred from the estimated joint relevance, which is based on the relevance w.r.t. the specific services; in fact, even if the TEO algorithm beats a vertical nested-loop by a factor of 2, the steps needed to retrieve all the results are still high. Indeed, results are centered in a small area far from the origin of the axes, and in order to better approximate optimality one should detect such "locally promising areas" and heuristically try to maintain fixed for a while one of the parameters (a single element or a batch of elements, depending on the cost model) so as to try to match it with all the results obtained from the other service - or at least to keep trying until the "local enhancement" seems to have terminated. In order to take advantage of such opportunities, our framework can be extended as follows.

![Figure 12: Comparison of actual vs expected scores](image)

Generally speaking, extraction-optimality builds on the assumption that, given two items \(x_i\) and \(y_j\) and their combination \(r_k\), the product of their ranking scores \(\rho_k \times \rho_k\) of \(r_k\), as stated in section 2.4. This is true only when the correlation of \(\rho_k\) and \(\rho_k\) is strong. Often, the correlation between these variables is weak, so it happens that the expected relevance \(\rho_k\) is different from the actual relevance \(\rho_k\). For instance, Figure 12 compares the trends of the expected and actual relevances for increasing values of the visited tiles, as observed in a given experiment. The figure illustrates several fluctuations in the actual relevance, while the expected relevance decreases linearly. These fluctuations are due to items with the same characteristics described in the previous example.

### 4.4 Runtime heuristics and adjustments

In this section, we sketch an heuristics that changes the navigation strategy dynamically, by taking into account the actual \(\rho_k\) of the combined results returned
by the matching operations. Actual $\rho_k$ values can sometimes be computed, because an exact test can be done about the “matching” of each result entry with the query; in other cases, they can easily be specified by users, who empirically judge whether an entry belongs to the query result (e.g., by clicking on predefined options in the user interface); coherently, run-time heuristics may take advantage of the user’s intervention.

In order to produce results more efficiently, several heuristic strategies can be used to try to take advantage of the positive fluctuations. When such a fluctuation is observed, a greedy algorithm explores the neighborhood of the last tile, where better results are likely to be found, disregarding the tile that would be chosen by the “static” algorithm.

An example of greedy heuristic is described in Figure 13, as a variant of the TEO algorithm. Assume that a positive fluctuation is observed in the tile marked with a white X (1); then, the exploration of the tile immediately below X is suggested (2), and this strategy continues, while the search produces positive results, until the border of the search space (3), where the next tile would cause a request-response. At this stage, another heuristic could suggest continuing the exploration of the (rectangular) search space (4); then, another heuristic would suggest continuing the search with the service $S_5$ \(^5\). We have implemented a greedy heuristic that, in presence of positive fluctuations, modifies the TEO algorithm as indicated in Figure 13, steps (1-4), and then resumes the static algorithm.

Figure 11 shows the improved performance of the heuristic algorithm over the standard Tile Extraction Optimal (rTEO stands for “reinforced TEO”).

---

\(^5\)These search strategies resemble the tactics of sailors during a race, when the race zone is explored looking for the “good wind”, and this can come “from the right” ($S_r$) or “from the left” ($S_l$); every positive fluctuation is a blow of the wind.

5 Related works

Enhancing search engines is a known research focus since a long time, well represented in the Asilomar report [7]; a summary of hard Web search problems is in [25]. We overview three research fields which constitute the premises to our work: search engines, Web Services, and XML-based string matching.

**Search Engines and Information Retrieval**

Search Engines, among the most sophisticated and useful resources available on the Internet, assist the user in the task of rapidly and effectively navigating the Web. To some extent, the problem of finding information on the Web can be rephrased as the problem of knowing where search engines are, what they are designed to retrieve, and how to use them. Two different types of engines have been developed so far: large-scale and specific search engines. Large-scale engines exemplify the trade-off between breadth and quality, while the specific ones are more likely to quickly focus a search in one particular area.

Information Retrieval systems are software tools which help users in the task of finding documents contained in a specific corpus or database. Such systems are also widely used on the Web for finding scholarly information as well as for many other recreational activities. The most popular information retrieval technique involves combining the full text of all documents within a corpus into an inverted index. Search engines use the common tf-idf ranking algorithm (term frequency times inverse document frequency), to exploit two important qualities of natural-language text to perform accurate retrieval: term frequency and inverse document frequency [26].

There have been few studies comparing the retrieval results of different search engines using different query formulations [13, 12, 16]. [13] presents comparisons using unrelated queries. Lucas and Topi [20] use eight search topics from which naive and expert queries were formulated and submitted to various Web search engines to evaluate relevancy. [12] explores the precision of search engines using a variety of topics and query formulations, noting that precision did not necessarily improve with the use of the advanced query operators.

The use of previous schema and mapping knowledge has been proposed in the past, but in two very restricted settings. They either use previous mappings to map multiple data sources to a single known mediated schema [11] or compose known mappings to a common schema [10].

Since search engines dole out a limited number of clicks per unit time among a large number of pages, always listing highly popular pages at the top, and because users usually focus their attention on the top few results [17, 18], newly-created but high-quality pages are “shut out”.

**Web Services**

The basic language for describing Web Services is
WSDL [22]. The composition of Web Services to constitute complex conversations is not supported by WSDL, but is the main purpose of several proposals, the most recent of which are BPEL4WS [3], WSCI [23], and WSCL [24]. An example of formalization of Web Services is offered by [6]; the article describes a framework defining a set of elementary actions, and a set of Web Services whose implementation rely only on these actions. [14] gives a broader view on the Web Services scenario, by using contributions from the theory of computation to assess the Web Service properties that can be automatically inferred. [19] describes the design and implementation of a service matchmaking prototype which uses a DAML-S based ontology and a Description Logic reasoner to compare ontology based service descriptions. [21] describes a technique for the semantic matching of Web Services.

6 Conclusions

In this paper, we have studied the new problem of join- ing the results of heterogeneous search engines. Our prototype has been able to “locate ethnic restaurants surrounding a given city”, that was regarded as a difficult and relevant problem. While the current prototype covers the design and implementation of join methods for search engines, i.e. the hard problem to be solved, we will next complete the development of the search engine integration framework, by developing the missing software modules. We also plan to define new join methods, based either on more sophisticated heuristics, or on the interplay with the human intervention.

References