

Optimization of Multi-Domain Queries on the Web

Daniele Braga Stefano Ceri Florian Daniel Davide Martinenghi

Dipartimento di Elettronica e Informazione – Politecnico di Milano
Piazza Leonardo da Vinci, 32 – 20133 Milano, Italy

{braga,ceri,daniel,martinen}@elet.polimi.it

ABSTRACT

Where can I attend an interesting database workshop close to a sunny beach? Who are the strongest experts on service computing based upon their recent publication record and accepted European projects? Can I spend an April weekend in a city served by a low-cost direct flight from Milano offering a Mahler's symphony? We regard the above queries as multi-domain queries, i.e., queries that can be answered by combining knowledge from two or more domains (such as: seaside locations, flights, publications, accepted projects, conference offerings, and so on). This information is available on the Web, but no general-purpose software system can accept the above queries nor compute the answer. At the most, dedicated systems support specific multi-domain compositions (e.g., Google-local locates information such as restaurants and hotels upon geographic maps).

This paper presents an overall framework for multi-domain queries on the Web. We address the following problems: (a) expressing multi-domain queries with an abstract formalism, (b) separating the treatment of "search" services within the model, by highlighting their differences from "exact" Web services, (c) explaining how the same query can be mapped to multiple "query plans", i.e., a well-defined scheduling of service invocations, possibly in parallel, which complies with their access limitations and preserves the ranking order in which search services return results; (d) introducing cross-domain joins as first-class operation within plans; (e) evaluating the query plans against several cost metrics so as to choose the most promising one for execution. This framework adapts to a variety of application contexts, ranging from end-user-oriented mash-up scenarios up to complex application integration scenarios.

1. INTRODUCTION

The current evolution of the Web is characterized by an increasing availability of online services (e.g. book search services provided by online stores or libraries) and novel search facilities (e.g. flight search Web sites, provided by

most commercial airlines or travel offers integrators). Being specific to a restricted domain, they offer a quality of their answers that goes much beyond what can be achieved via conventional, general purpose search engines. The overall amount of data that can contribute to such queries is continuously growing, mainly within the so-called *deep Web* [2], i.e., in a form not immediately indexable by search engines.

In light of these considerations, multi-domain queries as the ones mentioned in the abstract no longer represent a mere academic exercise; rather, they witness how *intricate* real life queries may be, and what a user would like to find available in order to fulfill real needs. However, we are still lacking effective query systems on the Web allowing users even to ask similar queries. General purpose search engines fail to answer multi-domain queries, while domain-specific search services cover a subset of the query domains. Thus, at the current state-of-the-art, the above queries can be answered only by patient and expert users, whose strategy is to interact with specialized services one-at-a-time and then feed the result of one search as input to another, reconstructing answers in their mind.

This paper delves into the research issues that arise in developing and optimizing a query system for multiple-domain Web queries, focusing on the specific features due to the presence, among them, of several search services. The distinguishing feature of a search service is to return answers in relevance order. In general, although the answers produced by a search service can be very numerous, users are only concerned with the answers provided within the first batch. Thus, a query strategy that retrieves all the answers from a search service is not appropriate. On the other hand, only the user can correctly evaluate the relevance of answers produced by search engines; therefore, if a query involves several searches, all answers produced by the involved search engines should be composed in the query output and presented to the user for a correct evaluation. Moreover, the user expects results in ranking order; thus, while composing answers from multiple search services, answers should be presented according to a global ranking that is a good composition of the various partial rankings; in this way, the user can control the interaction, seeing "good answers", and behave by arresting the search, changing the input values, or even interacting with the environment according to a more complex protocol, aware of which services are being invoked¹.

The paper by Srivastava et al. [16], which introduced the

¹This is outside the scope of this paper but part of our research.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.

VLDB '08, August 24-30, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 000-0-00000-000-0/00/00.

notion of Web Service Management System, can be considered as the direct predecessor of this paper; however, [16] did not consider the difference between search services and other services, or the problem of ranking results in the answer. Thus, our model and results are different from our predecessor.

1.1 Contributions of the paper

We define an original *formal model* for the optimization and the execution of multi-domain queries over heterogeneous Web information sources, which serves as a unifying perspective for several diverse settings, ranging from user-oriented service mashups to vertical service integration frameworks. The originality of the model resides in introducing a simple and yet effective classification of services: *exact* services have a “relational” behavior and return either a single answer or a set of unranked answers, *search* services return a list of answers in ranking order, according to some measure of relevance.

We formally define *query plans* as equivalent to relational physical access plans; query plans schedule the invocations of Web services and the composition of their inputs and outputs. A plan is defined as the orchestration of service invocations, possibly in parallel, which takes into account the most significant features of the service, including its ability to page the results (i.e., to return a given number of answers with a single request-response). Within plans, the main operations are joins between Web service results, whose execution can take place according to several join strategies.

We define an *optimization technique* resorting to a classical and well-grounded approach such branch and bound in order to efficiently explore the solution space and find the optimal execution plan. Cost minimization is performed according to one of several alternative *cost metrics* capturing different scenarios.

We give *experimental evidence* that our proposed model fits well on a set of services and Web information sources that allow accessing deep Web data. We specifically focus on immediately invocable Web services and manually and semi-automatically generated wrappers for data intensive Web sites.

The approach and the techniques described in this paper are core components of a multi-center Italian research project (*New Technologies and Tools for the Integration of Web Search Services*) [3].

2. OVERVIEW OF OUR APPROACH

The optimization problem considered in this paper is: *given a query over a set of services, find the query plan that minimizes the expected execution cost according to a given metric in order to obtain the best k answers.* The process of generating an optimal plan starts from a datalog-like formulation of the query and ends with a fully instantiated invocation schedule.

The execution environment is a system capable of executing query plans (as they will be formally defined in the sequel); this means that the system can execute Web Service requests, collect their results, and integrate them progressively, forming the “answers”.

2.1 Characteristics of the information sources

We consider conjunctive queries (i.e., select-project-join queries) on services over information sources available on the Web. We abstract away from the technicalities of the underlying data representations and resort to simple signatures with a name and a list of attributes for each source.

A fundamental distinction in our model concerns the nature of services; we distinguish between exact and search services. *Search* services return a list of tuples in ranking order, according to some measure of relevance; such measure is normally opaque (i.e., not visible in the result). *Exact* services return either a single tuple or a set of unranked tuples; these tuples cannot be distinguished with respect to a preference criterion.

It should be noted that Web sources are not freely accessible as in the traditional relational setting, because they typically expose a limited number of interfaces, in which certain fields must be mandatorily filled in order to obtain a result. These fields may be the input fields of a form on a data-intensive Web site or the input parameters of a Web Service invocation. Such access limitations are crucial to the optimization problem, and modeled as follows. We assume that each service is characterized by a given number of combinations of input and output parameters, corresponding to the different ways in which it can be invoked. Along with the literature, we call each such combination an *access pattern* for the service. Intuitively, a query including several services is *executable* if a schedule exists that allows invoking each service mentioned in the query according to an available access pattern. This may occur either because the user has specified all the input values required for the input fields or because values output by a service can feed other services’ input fields, so as to globally comply with the available access patterns.

Disregarding the access patterns, services can be abstractly thought of as relations. In general, each service has an associated *cardinality*, expressing the total number of tuples that it could produce after being called in all possible ways. However, to our purposes, it is more important to characterize a service through its *expected result size per invocation* (erspi), i.e., the average number of results produced by one invocation. If a service’s erspi is greater than 1 we say, along with [16], that the invocation is *proliferative*; if it is between 0 and 1 we say that it is *selective*. Normally, search services are highly proliferative, thus the retrieval of their tuples must be halted, while exact services can either be selective or proliferative.

Services can further be classified as “*bulk*” or “*chunked*”; in the former case, they return all their results as effect of a single request, in the latter case they return results in chunks (or pages) of a fixed size; this occurs in general with search services, and can occur whenever a service returns a large number of results. In our abstract model, we characterize chunked service with a *chunksize*, which expresses the number of tuples returned by each “fetch” (i.e., sequential invocation) performed on the service.

Joins can be considered as particular exact services provided by the system, capable of merging results from two services if they satisfy a join condition. Also, joins have an associated erspi; the erspi of a join over two services is given by the product of the erspi values of the services multiplied by the selectivity of the join condition.

2.2 Query plans

A *query plan* indicates the sequence of invocations of services and their conjunctive composition through joins. The specification of a query plan allows the execution of a query as a dataflow computation, from the user’s input to the first k answers. We represent plans as directed acyclic graphs (DAGs):

- Every node represents either an atom in the conjunctive query (i.e., a service invocation) or a join. Every node is characterized by an erspi value, chunked atoms are further characterized by a chunksize value.
- Every arc indicates a precedence in the invocation and possibly a parameter passing from values of the results of one service to inputs of another service.
- Every join node is marked with an indication of the join strategy to be employed, and an indication of the number of fetches to be performed on each joined service, if they are chunked.

The graphical syntax that we use for representing query plans will be introduced in detail in Section 3.3 (Figure 4).

Constant values appearing in a query are either presented by the user through a form or set within a query template; optimization is performed for each query template under suitable assumptions of domain uniformity and independence. In our framework, we retrieve only the fraction of tuples of proliferative services that are sufficient to obtain the first k query answers; we set k so that the answered tuples should normally satisfy the user’s needs, but we also assume that a plan execution can be continued, by producing more answers. A user can either be satisfied with the first k answers, or ask for more results of the same query, or change the choice of keywords and resubmit a new query with the same template, or turn to a different Web activity. We assume that services are independent of each other and that at each service call the values are uniformly distributed over the domains associated to their input and output fields. These assumptions allow us to obtain estimates for predicate selectivities and sizes of results returned by each service call; cost models use erspi and chunksize parameters.

2.3 Cost Metrics

A *cost metric* is a function that associates a cost to each query plan. We mainly consider the following cost metrics:

- *Sum cost metric*, which computes the cost of the plan for producing k answers as the sum of the costs of each operator used in the plan. Examples of costs for a service invocation are the cost of computing joins or the cost charged by the service.
- *Execution time metric*, which measures the (expected) time elapsed from the query submission time to the production of the k -th answer.

A special case of the sum cost metric is the *request-response cost metric*, which consists of counting the number of service invocations required to execute the plan; it assumes that service invocations are the only relevant operations and their cost is set to 1. This metric is particularly relevant when the transfer of data over the network is the dominating cost factor.

In our examples, we will use the execution time and request-response metrics, as they best captures the typical usage

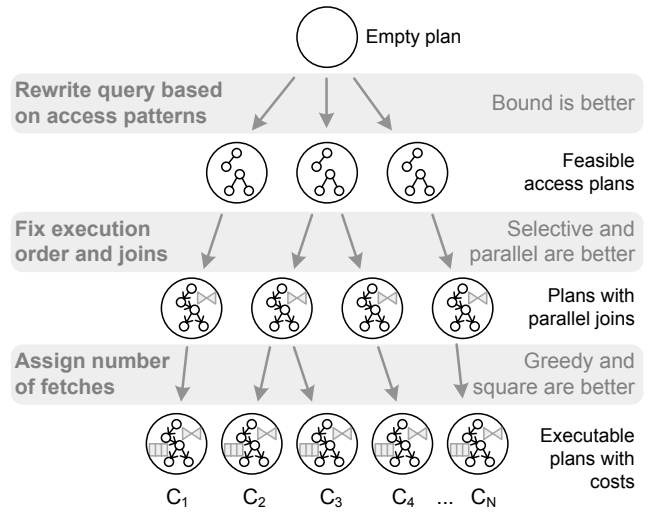


Figure 1: Overview of the global optimization phases

scenario we have in mind, where a query is executed in an open world of free-of-charge services, and where determining fine granularity service execution costs is difficult.

There are other cost metrics of interest, though not considered in detail in the rest of the paper:

- *Bottleneck cost metric*, which gives the execution time of the slowest service in the plan, and is relevant in contexts of pipelined execution of continuous queries. This metric, fully studied in [16], is suitable to contexts with homogeneous services (resembling a distributed DBMS) but it is not advised in our context, where search services rarely produce all their tuples and the execution is normally limited to reaching k answers.
- *Time-to-screen cost metric*, which measures of the time required to present the user with the first output tuple, accounting for settings in which the user expects a prompt interaction.

2.4 Optimization Approach

Finally, after characterizing services, query plans, and cost metrics, we are in the position of introducing our overall optimization approach; this is the paper’s main contribution, represented in Figure 1. The approach consists in exploring a highly combinatorial solution space that characterizes all possible translations from the user query into fully instantiated query plans. The exploration is separated into three phases, that provide subsequent details about query plans.

- The **first phase** is the *selection of a given query rewriting* such that every service is called with one of the available access patterns. This phase transforms the conjunctive query over web services into several annotated access queries over access patterns of the corresponding services.
- The **second phase** is the *selection of a query plan* for the given query rewriting. This phase fixes the order of execution of the query over the services as well as the position and kind of joins between services used in the plan.

```

conf(1)(Topic, Name, Start, End, City)
conf(2)(Topic, Name, Start, End, City)
weather(City, Temperature, Date)
flightS(From, To, OutDate, RetDate, OutTime, RetTime, Price)
hotel(1)S(Name, City, Category, CheckInDate, CheckOutDate, Price)
hotel(2)S(Name, City, Category, CheckInDate, CheckOutDate, Price)

```

Figure 2: Schema of the available services

- The **third phase** is the *assignments of the exact number of fetches* to be performed over the chunked services. This phase allows to fully determine the execution strategy for a query and therefore to compute its cost according to a given metrics.

Each phase is combinatorial, hence the considered problem is intractable by exact methods, even with simple queries. However, all considered cost metrics are monotonic, which suggests an exploration of the space with a *branch and bound* strategy.

Each choice in a construction step corresponds to the selection within a class of DAGs (i.e., those that may be constructed starting from the initial query or the previous choice); all possible classes at a given step determine a subdivision of the search space into non-overlapping subsets, which is an ideal *branching*. Then, thanks to the mentioned monotonicity, each DAG of a class can be assigned a *lower bound* for the cost by calculating the cost on the partially constructed DAG. To complete the *bounding* step, we can obtain an *upper bound* for a class of DAGs by fully constructing one DAG in the class and calculating its cost. With this, we may apply the *pruning step*: if the lower bound for some class A is greater than the upper bound for some other class B , then A may be safely discarded from the search.

This approach is used within database optimizers; e.g. the analogous phases in join optimization consist first in determining the join order, then the join method, then its parameterization according to the supported join execution procedures. The considered problem has a similar combinatorial explosion, hence there are good hopes that optimizers could find sufficiently good solutions in acceptable computation time.

2.5 Running example

Consider the query “find all database conferences in the next six months in locations where the average temperature is 28°C degrees and for which a cheap travel solution including a luxury accommodation exists”; answering this query requires: (i) finding interesting conferences in the desired timeframe via online services by the scientific community; (ii) understanding whether the conference location is served by low-cost flights; (iii) finding luxury hotels close to the conference location with available rooms; and (iv) checking the expected average temperature of the location.

This query will be walked through in the next sections.

2.6 Structure of the paper

Section 3 formally defines the three models. Then, Section 4 defines the constituting elements of the branch and bound method, by assigning to each phase both a suitable *first choice heuristics* (a good first choice is essential for building an effective upper bound) and a suitable *exploration strategy* for exhausting all alternatives in the search space. In Section 5 we discuss cost metrics in further detail, and give some re-

```

q(Conf, City, HPrice, FPrice, Start, StartTime, End, EndTime, Hotel) :-
flight('Milano', City, Start, End, StartTime, EndTime, FPrice),
hotel(Hotel, City, 'luxury', Start, End, HPrice),
conf('DB', Conf, Start, End, City),
weather(City, Temperature, Start),
Start >= '2007/3/14', End <= '2007/3/14' + 180,
Temperature >= 28, FPrice+HPrice < 2000.

```

Figure 3: Query from the running example

sults that hold under simplifying assumptions either related to the metrics or to the considered problem (specifically, the number of involved search services). We next present experiments validating our approach and related work.

3. FORMAL MODEL

3.1 Notation for queries and services

Formally, each service s is equipped with a signature of the form $s^\alpha(A_1, \dots, A_n)$, where s is the service name, n is called the *arity* of the service, each A_i is an abstract domain², and α is a set of feasible access patterns for s . An *access pattern* α is a sequence of ‘i’ and ‘o’ symbols of length n ; for $1 \leq k \leq n$, the k -th argument of s is said to be an *input* argument in α if the k -th symbol in α is ‘i’, an *output* argument otherwise. A *schema* is a set of signatures for different services.

We denote variables by uppercase letters and constants by lowercase letters, numbers or as strings enclosed in quotes; variables and constants are collectively called *terms*. An *atom* for a schema \mathcal{S} is an expression of the form $s(t_1, \dots, t_n)$, where s is the name of a service with a signature of the form $s^\alpha(A_1, \dots, A_n)$ in \mathcal{S} , and each t_i is a term; if t_i is a constant, then it belongs to the abstract domain A_i . For convenience of notation, we sometimes indicate a sequence of terms (or other objects) t_1, \dots, t_n as \bar{t} and a tuple $\langle c_1, \dots, c_m \rangle$ as $\langle \bar{c} \rangle$; the length of a sequence \bar{t} is denoted by $|\bar{t}|$.

We use a datalog-like notation for queries. A *conjunctive query* (CQ) q of arity n over a schema \mathcal{S} is written as

$$q(\bar{X}) \leftarrow \text{conj}(\bar{X}, \bar{Y})$$

where $|\bar{X}| = n$, $q(\bar{X})$ is called the *head* of q , $\text{conj}(\bar{X}, \bar{Y})$ is called the *body* of q and is a conjunction of comma-separated atoms for \mathcal{S} involving the variables in \bar{X} and \bar{Y} and possibly some constants. We assume that queries are *safe*, i.e., that each variable of the query appears in at least one atom in the body. Note that CQs with atoms corresponding to different services represent multi-domain queries.

Given a schema \mathcal{S} , the *answer* q^D to a CQ q over \mathcal{S} with data instance D is the set of tuples $\langle \bar{c} \rangle$ of constants, with $|\bar{c}| = |\bar{X}|$, such that there is a sequence of constants \bar{d} , with $|\bar{d}| = |\bar{Y}|$, for which each atom in $\text{conj}(\bar{c}, \bar{d})$ is in D .

For uniformity of notation, we shall always use the same letters to refer to service properties. In particular, ξ_s will indicate the erspi of a service s and τ_s its average response time. If s is chunked, cs_s will indicate its chunksize and F_s the number of performed fetches; if s is a search service, ds will indicate its decay, i.e., the number of tuples after which ranking is known to decrease under the threshold of interest (if such information is available). For a selection predicate p , we indicate its selectivity as σ_p .

²Note that we use a positional notation and that the A_i ’s do not denote attributes but abstract domains.

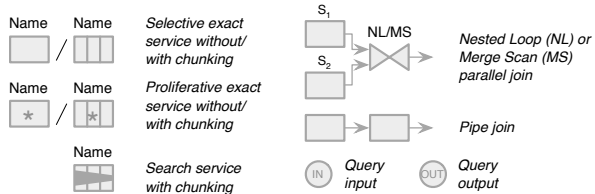


Figure 4: Visual syntax for representing query plans

EXAMPLE 3.1. The schema of our running example is represented in Figure 2, where, for ease of notation, we have rewritten a signature for each access pattern, by adding a different subscript index to the service name, and by underlining the input fields. For instance, the `conf` service would have the signature $\text{conf}^{i0000,0000i}$ (Topic, Name, Start, End, City), according to the notation introduced previously. Search services (extracting hotels and flights in ranking order) are indicated with an “S” superscript that will be omitted in the following. The query from our running example is shown in Figure 3. ■

3.2 Access patterns

We consider a conjunctive query expressed over a set of services $\{s_1, \dots, s_n\}$ with access patterns; we assume that each service s_i has m_i feasible access patterns, for $1 \leq i \leq n$. Clearly, there are $\prod_{1 \leq i \leq n} m_i^{o_i}$ possible ways of selecting an access pattern for each atom in the query body, where o_i is the number of occurrences of service s_i in the query.

Although the number of choices is potentially very high, typically, only few of these lead to an executable query, as specified next.

DEFINITION 3.1. Let Q be a conjunctive query of the form $H \leftarrow B_1, \dots, B_n$. An atom B_i is callable in Q with respect to a sequence $\bar{\alpha} = \langle \alpha_1, \dots, \alpha_n \rangle$, where each α_i is a feasible access pattern for B_i 's service, for $1 \leq i \leq n$, if

- each of its input fields is filled with a constant, or,
- inductively, each of its input fields is filled with a constant or contains a variable that occurs in an output field of a callable atom.

Q is executable with respect to $\bar{\alpha}$ if each B_i is callable, for $1 \leq i \leq n$; in this case, we say that $\bar{\alpha}$ is permissible in Q .

In [21] the authors propose an algorithm to check the existence of a permissible sequence of access patterns for a conjunctive query; the time complexity of this algorithm is linear in the size of the query³. Non-permissible sequences are therefore discarded from consideration at the very early stages of our approach.

3.3 Query Plans

Once a permissible sequence of access patterns is chosen, the variables in the query determine precedences between service invocations. For instance, whenever the same variable X occurs both in an output field of an atom A_1 and in

³Under the reasonable assumption that the number of feasible access patterns for each service is bounded by a constant.

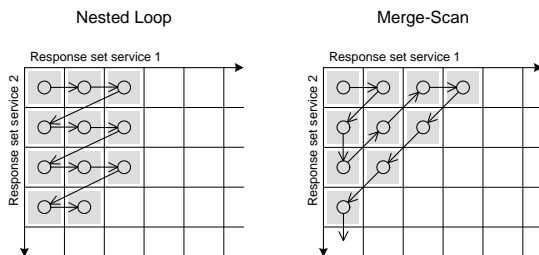


Figure 5: Join strategies: nested loop and merge-scan

an input field of an atom A_2 , and nowhere else, this indicates that A_1 's service must be called before A_2 's service for X to provide a binding for A_2 's input. In other words, A_1 necessarily precedes A_2 during the execution of the query; we write $A_1 \prec A_2$.

More generally, precedences between atoms determine a partial order on the query atoms. Following the intuition of Definition 3.1, we say that an atom A is callable after N , where N is a set of query atoms, if $A \notin N$ and A 's input fields contain a constant or a variable occurring in an atom in N ; an atom is directly callable if it is callable after \emptyset – obviously, atoms with no input field are directly callable. The set of atoms callable after N in a query Q is denoted $\text{callable}_Q(N)$.

A query plan is represented by a DAG that complies with the precedences between atoms, i.e., if N is the set of nodes (corresponding to the query atoms A) preceding a node N (corresponding to the query atom A) in the DAG, then A must be callable after A . From now on, we will use the terms node and atom interchangeably.

Placing a node on the DAG means representing the invocation of the corresponding service. If two nodes are connected by an arc in the DAG, the destination is invoked after the origin; if they are not connected by any directed path, they are invoked in parallel.

Figure 4 introduces the graphical modeling notation for DAGs. A query plan has a unique start node (the user query's input) and a unique end node (the query result). Selective, exact services are represented as simple boxes; proliferative, exact services are represented as boxes labeled with an asterisk; search services are represented as boxes with a grey trapezium (sketchily representing the decrease in ranking of the results). Chunked services are represented by splitting the service's box into three smaller boxes. As for joins, we distinguish between two join patterns: *pipe join* and *parallel join*. 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. Indeed, this way of joining the results corresponds to a feed-forward of values. Instead, a parallel join occurs when join predicates are applied to fields that are in output for both involved services.

Parallel joins enable parallelizing the use of Web services and thus are fundamental operations of query plans. They are represented by means of dedicated nodes, shaped as join symbols, with an associated label expressing the respective join method (denoted as NL to mean “nested loop” or MS to mean “merge-scan”). These methods are the subject of a dedicated article [4], restricted to binary joins, whose main results are summarized below for the reader's convenience.

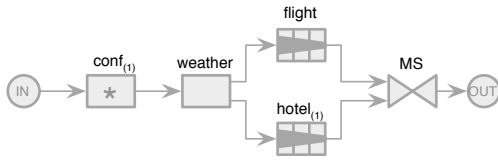


Figure 6: A query plan for the running example

If we represent the items returned by the two involved services on two Cartesian axes, each point in the plane represents a candidate join result, to be tested against the join condition. Along with Figure 5 we can represent the join strategies as ways of scanning such search space.

- NL is employed when there is one service that is highly selective, and produces the highly ranked tuples with few fetches; in this case, the result space is explored by executing first the (few) fetches of the selective service, then the (many) fetches of the other service, and immediately scanning the search space while the tuples of the less selective field become available.
- MS is employed when there is no a priori distinction between the selectivity of 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”.

In both cases, the traversing strategy outputs tuples in a global order that is consistent with the partial orders of each service. In general, the choice between NL or MS depends on the particular pair of services and can be made at service registration time, by analyzing their statistical behavior. An example of possible query plan at this stage is shown in the DAG of Figure 6.

3.4 Annotated Query Plans

For each node n in the plan we can estimate the number of tuples in output, denoted as t_n^{out} . By t_n^{out} we mean the sum of all outputs of all invocations of the service associated to n . We assume that the user always injects one single input tuple in the plan, represented by the start node. For exact services, t_n^{out} is given by the product of t_n^{in} (the number of input tuples, each of which a priori requires one invocation of n) by ξ_n , the erspi of n . For chunked services, t_n^{out} equals $cs_n \cdot F_n$ (where F_n is the number of fetches). If node n represents a join (say, of the outputs of nodes l and m), t_n^{out} is given by the product $t_l^{out} \cdot t_m^{out} \cdot \sigma_p$, where σ_p is the selectivity of the join predicate p . The selection predicates applied to all service invocations are included for convenience in the notion of erspi. Estimating the erspi of a service does not differ, in principle, from what is normally done to estimate the effect of a selection predicate over a table in a relational database.

The overall result size of the query, denoted as t^{out} , only depends on the number of fetches F_{n_i} for all chunked services n_i . These are the only open parameters fixed in this phase. An annotated plan with the values F_{n_i} of its chunked services is executable, and thus can be associated with an execution cost. Figure 8 shows an example that also includes

the expected size of intermediate results; it would produce enough answers with, e.g., k set to 10. The calculations needed to obtain such values are discussed in Section 5.3.

4. INSTRUMENTING THE BRANCH AND BOUND METHOD

This section addresses the three optimization phases introduced in 2.4. For each of these three phases (choice of access patterns, topology of the plan, and number of fetches) we give some heuristics for choosing an initial choice and a definition of the branching and bounding steps to be used in the global optimization process for examining the other solutions in the space. The heuristics perform first assignments of the open parameters, so as to provide the branch-and-bound strategy with a heuristically good upper bound to start the exploration of the search space and rapidly achieve convergence. An intuitive snapshot of the process has been given by Figure 1.

4.1 Access Pattern Selection

4.1.1 Heuristics: bound is better

A good heuristics for selecting the most promising choices of access patterns among the permissible ones consists in preferring access patterns with input fields wherever possible; we refer to this heuristics as “bound is better”.

More formally, given two feasible access patterns α_1 and α_2 for a service s , we say that α_1 is *more cogent* than α_2 , written $\alpha_1 \preceq_{IO} \alpha_2$, if every field in s that is marked as input in α_2 is also marked as input in α_1 .

Generalizing, given two sequences of n feasible access patterns $\bar{\alpha}_1$ and $\bar{\alpha}_2$, for n corresponding services, we write $\bar{\alpha}_1 \preceq_{IO} \bar{\alpha}_2$ if $\bar{\alpha}_1[i] \preceq_{IO} \bar{\alpha}_2[i]$ for $1 \leq i \leq n$, where we have indicated with $\bar{\alpha}[i]$ the i -th element of a sequence $\bar{\alpha}$. We write $\bar{\alpha}_1 \prec_{IO} \bar{\alpha}_2$ whenever $\bar{\alpha}_1 \preceq_{IO} \bar{\alpha}_2$ and not $\bar{\alpha}_2 \preceq_{IO} \bar{\alpha}_1$. A sequence $\bar{\alpha}$ of n feasible access patterns is *most cogent* whenever there is no other sequence $\bar{\alpha}'$ of n feasible access patterns such that $\bar{\alpha}' \prec_{IO} \bar{\alpha}$. Any most cogent choice of access patterns is a heuristically good initialization. This choice is grounded on the following considerations.

- Given a fixed cardinality for a service, an invocation with a more cogent access pattern is much more likely to return a smaller answer set, and certainly cannot return a bigger one. Therefore, fewer requests are needed to retrieve the data, and the size of intermediate results is heuristically reduced. In analogy with traditional databases, this corresponds to pushing selections near to data sources.
- If the service is implemented with suitable indices on the bound fields, a more cogent service is more likely to respond more quickly.
- With smaller intermediate results, less computational effort is required for caching data and for operating on them.

These observations hold for all the cost metrics that we consider, which therefore all benefit from the “input is better” heuristics.

4.1.2 Exploration of the search space

This phase explores the space of permissible sequences of patterns, starting with the most cogent choices, and continuing with the other choices. Lower bounds for the patterns can be computed by isolating the services that are less cogent than some services in an already computed solution, and then by computing the cost associated to those services under the most favorable assumptions; the bound is effective if such cost exceeds the complete cost of the considered solution.

EXAMPLE 4.1. According to the feasible binding patterns, there are 4 possible choices corresponding to the services `conf`, `flight`, `hotel`, `weather` used in the query of Example 3.1:

$$\begin{aligned}\bar{\alpha}_1 &= \langle \text{conf}_1, \text{flight}, \text{hotel}_1, \text{weather} \rangle \\ \bar{\alpha}_2 &= \langle \text{conf}_1, \text{flight}, \text{hotel}_2, \text{weather} \rangle \\ \bar{\alpha}_3 &= \langle \text{conf}_2, \text{flight}, \text{hotel}_1, \text{weather} \rangle \\ \bar{\alpha}_4 &= \langle \text{conf}_2, \text{flight}, \text{hotel}_2, \text{weather} \rangle\end{aligned}$$

However, $\bar{\alpha}_3$ is not permissible in Q , since variable C in Q then always occurs in input fields, and therefore none of the query atoms is callable. In addition, we have $\bar{\alpha}_1 \prec_{IO} \bar{\alpha}_2$, since `hotel`₂ only has output fields. Therefore the only two most cogent choices are $\bar{\alpha}_1$ and $\bar{\alpha}_4$. ■

4.2 Query Plan Selection

4.2.1 Heuristics: selective and parallel are better

Two independent heuristics for obtaining a good upper bound are: (i) having one single path in the DAG, ordered by increasing `ersp`i wherever possible, or (ii) always making the choice that maximizes parallelism; of course, this does not necessarily minimize the cost. Generally speaking, incrementing the parallelism plays in favor of those metrics that take time into account, while sequencing selective services plays in favor of metrics that minimize the overall number of invocations. In absence of access limitations, this gives the optimal solution, as proved in [16].

4.2.2 Exploration of the search space

The construction of all possible DAGs for a query Q takes place incrementally. It starts by placing after the initial node some directly callable nodes, and then by progressively adding nodes that are callable after the placed nodes. The notion of (directly) callable was introduced in 3.3.

The plan can start with any single directly callable node, or with the parallel of any number of such nodes. Once a set of nodes N_i is placed on the DAG, one can add the parallel of any subset N_{i+1} of the nodes in $\text{callable}_Q(N_i)$ such that

- any node in N_{i+1} has an incoming arc originating from a node in N_i placed during the previous step, and
- there is an arc (A, B) for any node $B \in N_{i+1}$ such that $A \prec B$.

In other words, at each step of the construction, the number of choices for placing a node on the DAG equals $|\text{callable}_Q(N_i)| - 1$, i.e., the cardinality of the power set of the placed nodes minus 1; for the first step, there are $|2^{N_1}| - 1$ choices, where N_1 is the set of directly callable atoms.

Clearly, the space of constructible DAGs may grow very quickly, due to the exponential number of choices at each

step of the construction (yet, the choices depend on the degrees of freedom on the partial order induced by the access patterns – if they determine a total order, then there is only one possible DAG).

4.3 Chunked Service Selection

Whenever a query includes chunked services, we need to provide an estimate of the number of chunks that will be retrieved per input tuple at the service. This number is the *fetching factor* of the service, and we call *fetch* the operation for requesting a chunk of results.

4.3.1 Heuristics: greedy and square are better

We consider two possible heuristics.

- “*Greedy*”. Initially all fetching factors are set to 1, which is the lowest admissible value for such parameters. This is already the optimal solution if the number h of tuples in the result already equals or exceeds k . Otherwise, we iteratively increment the fetching factor of the node with the highest sensitivity with respect to the increase in the number of tuples per cost unit; we stop as soon as $h \geq k$. This procedure finds a local optimum, which coincides with the global optimum if the search space is convex.
- “*Square is better*”. All fetching factors are initially set to 1, as in the greedy heuristics, but, at each iteration, each of them is incremented by a value that is proportional to its chunk size, until $h \geq k$. This implies that, in average, after query execution, all chunked services will have explored about the same number of tuples. By “democratically” assigning proportional fetching factors to all chunked services, this heuristics suits scenarios in which ranking of search services quickly decreases, and fetching many chunks of results only from few, selected services does not pay off.

4.3.2 Exploration of the search space

Clearly, if the n -tuple $\langle 1, 1, \dots, 1 \rangle$ determines $h \geq k$ results, then it is also the optimal solution. Otherwise an exploration of the search space is needed, starting from the initial assignment determined by the chosen heuristics. In general, given n chunked services, finding the best n -tuple of fetching factors is a combinatorial problem. Each fetching factor F_i ranges from 1 to some maximum integer value F_i^{max} , which can be calculated as the minimum value for F_i to obtain $h \geq k$ when all the other fetching factors are 1.

However, not all n -tuples need to be considered, since some may be dominated by already explored ones: an n -tuple is dominated by another if all its integers are greater than or equal to those in the corresponding positions, and therefore need not be considered. The space of n -tuples can then be exhaustively explored by starting from the initial assignment, and varying the n -tuple by incrementing and decrementing some fetching factors so that the new n -tuple is not dominated by any previously explored tuple and none of its fields exceeds its upper bound. Besides simple enumeration, the exploration can be done in several ways, for instance by *first iterative deepening* on the sum of the variations on all fields.

Note that a known decay for a service i also provides an upper bound for F_i : after $\frac{d_i}{cs_i}$ fetches no relevant data are returned by i . Small upper bounds determined by decays may sometimes even mean that k answers can never be reached.

5. EXECUTION SETTINGS AND COSTS

This Section is concerned with the instantiation of the optimization framework onto specific architectures and specific cost models. Thanks to the above choices, a generic search strategy of optimal query plan may be significantly improved. We assume that any compliant execution environment would support:

- *Service registration*, i.e. a process by means of which the services become known to the optimizer; the registration includes several features about each service, such as its signature and its patterns, and gives estimates (by sampling) of its erspi, average response time, and chunk values. The estimates are periodically updated, also taking advantage of subsequent invocations. For each pair of services, it is known which parallel join method should be used.
- *Service orchestration*, including the join methods and the mechanisms for composing the answers and presenting them to the user.
- *Multi-threading*, i.e. the ability to invoke services in parallel as distinct threads associated with the same query execution.

5.1 Logical caching

A relevant aspect of the execution engine is *logical caching*, i.e. the ability of caching result tuples from services corresponding to given input parameters; this aspect is very useful in order to avoid calls that were already executed during the query plan execution. We distinguish between three different settings:

- *no cache*: every call is repeated.
- *one-call cache*: the system recalls the last call to each service and its results; this is enough for avoiding the re-issuing of any immediate “second-call” with exactly the same input parameters.
- *optimal cache*: the system recalls parameter settings and results of all calls, thus the execution of a query plan globally issues a number of calls to each service that is equal to the cardinality of its inputs (i.e., of the different values presented in input to all calls).

Repeating service invocations using the same parameters in consecutive calls may occur frequently in a query plan, as a consequence of the presence of proliferative services. The *one-call cache* setting best trades the savings with the simplicity of implementation and tuning of the execution engine, and therefore best captures the scenario of multi-domain Web queries. In particular, we will show that such a limited caching mechanism can perceptibly improve performances.

5.2 Accurate estimate of required invocations

Measuring the cost of a plan \mathcal{G} requires estimating how many times each service is invoked. At this stage, the above different caching scenarios require different ways to calculate t_n^{in} , the numbers of tuples in input for each node n (each of which, a priori, may require one invocation of n).

The authors of [16] assume that \mathbf{t}^{out} always equals

$$\prod_{n_i \in \text{nodes}(\mathcal{G})} \xi_{n_i} \quad (1)$$

This corresponds to a *no-cache* setting.

However, with caching, the nodes that are guaranteed not to vary the input tuples for n should not be considered. Indeed, t_n^{out} is given by $t_n^{in} \cdot \xi_n$ or $cs_n \cdot F_n$ for exact and chunked services respectively. Generally speaking, t_n^{in} retroactively depends on the erspi and fetching factors of all nodes preceding n in \mathcal{G} . As several consecutive invocations of the same service in a short period of time are likely to return the same result, fully determined by the values in input, the number of *actually required* calls for a service n may be smaller than t_n^{in} .

In a given plan, the relative position of two service invocations may either be: (1) independent, i.e., in parallel, if no path in the plan includes them both, or (2) in a sequence, when such a path exists. Sequenced services may have *field dependencies*, if some output of the former service feeds some input fields of the latter; otherwise, they are field-independent⁴. It should not be forgotten that the use of the same variable in the query indicates an equi-join. Therefore, for each input variable X in n , the number of input *distinct* values for X cannot exceed the minimum erspi of the services having X in output multiplied by the number of tuples they receive in input.

By construction, during the execution of a query, all the tuples originating from a proliferative service are retrieved contiguously, and will therefore be contiguously sent forward in the plan preserving the same values for the input fields of the invocation of non-dependent services. When such “blocks” of uniform tuples arrive to a node n with no field dependency from the node that originated the blocks, one call to n will suffice to compose n ’s results with all the tuples of each block. Therefore, for each input variable X of n , we consider the node m in the plan that has X in output *and for which t_m^{out} is minimal*; the product of all such minimal values, for all input variables, conservatively estimates the maximum number of required invocations of n . To boil all this down to formulas and give a compact expression for the cost functions, we write as follows:

$$t_n^{in} = \prod_{m \in \mathcal{N}(n)} \xi_m t_m^{in}, \quad (2)$$

where $\mathcal{N}(n)$ is the set of nodes giving the minimal contribution to each input variable of n . Indicating with (n_j, n_i, n) a path from n_j to n passing through n_i (n_j and n_i may coincide, n_i and n cannot) such set is:

$$\mathcal{N}(n) = \bigcup_{X \in \text{InVars}(n)} \left\{ m \mid t_m^{out} = \min_{\substack{n_i: (n_j, n_i, n) \in \text{paths}(\mathcal{G}), \\ X \in \text{OutVars}(n_j)}} (t_{n_i}^{out}) \right\}$$

5.3 Details of the considered cost metrics

Given the notion of field-dependent nodes and a notation $m(n)$ for the individual cost of invocation for service n , the global cost associated to a plan according to the *sum cost metric* is then given by the sum of the costs incurred by each service invocation:

$$SCM(\mathcal{G}) = \sum_{n \in \text{nodes}(\mathcal{G})} m(n) \cdot t_n^{in} \quad (3)$$

⁴In other words, two services that are sequenced but field-independent happen to be invoked in series according to the execution schedule, but no real parameter passing happens between them.

With the *execution time metric*, the cost $ETM(\mathcal{G})$ must account for the slowest path flowing tuples from the user input to the output of the plan, and associates to each path an execution time that consists of two components: (i) a *bottleneck*, which is the node in which the product of invocation/fetches multiplied by the time-per-invocation is maximal, and (ii) the time needed to reach the bottleneck node with the first tuple (filling in the pipe) plus the time to reach the output from the bottleneck (emptying out the pipe):

$$ETM(\mathcal{G}) = \max_{P \in \text{paths}(\mathcal{G})} \left[\left(\max_{n \in \text{nodes}(P)} F_n \cdot t_n^{in} \cdot \tau_n \right) + \sum_{m \in \text{nodes}(P) \setminus n_{bn}} \tau_m \right] \quad (4)$$

where n_{bn} is the bottleneck node maximizing the first term.

It is worth noting that both these metrics are monotonic with respect to the way in which DAGs are constructed in our framework, and that the values for t_n^{in} can be calculated accordingly to any of the considered settings, yielding different values.

5.3.1 Fetching factors: interesting special cases

We now address the problem of assigning fetching factors according to the heuristics for one of the aforementioned cost metrics.

In case only *one* chunked service s_n is in \mathcal{G} , its F_n may be easily calculated. Indeed, \mathbf{t}^{out} is a function of F_n only, so it suffices to equal its expression with k (the desired result size) to obtain a value for F_n . This is easy if the contribution $F_n \cdot cs_{s_n}$ of tuples from n can be isolated in the expression of \mathbf{t}^{out} , which happens, e.g., if n is the last node in \mathcal{G} or if all the outputs of a node are used as inputs for the subsequent nodes. In the latter case, \mathbf{t}^{out} equals the expression (1), therefore we have:

$$\mathbf{t}^{out} = \prod_{i \in \text{nodes}(\mathcal{G}) \setminus \{n\}} \xi_i \cdot F_n \cdot cs_{s_n}. \quad (5)$$

The factor $\prod_{i \in \text{nodes}(\mathcal{G}) \setminus \{n\}} \xi_i$ equals the *bulk erspi* $\Xi(\mathcal{G}) = \prod_{i \in \text{bulkNodes}(\mathcal{G})} \xi_i$, which is the product of all *erspi* for the bulk services in the DAG. In order to obtain k tuples in output, the fetching factor for n should be $F_n = \lceil \frac{k}{\Xi(\mathcal{G}) \cdot cs_s} \rceil$; obviously, F_n depends on k .

When *two* nodes n_1 and n_2 corresponding to chunked services s_1 and s_2 are present in \mathcal{G} , \mathbf{t}^{out} depends in general both on F_{n_1} and F_{n_2} ; therefore, k fixes a class of pairs for these fetching factors. Under the same assumptions as Equation (5), we have here: $\mathbf{t}^{out} = \Xi(\mathcal{G}) \cdot F_{n_1} \cdot cs_{s_1} \cdot F_{n_2} \cdot cs_{s_2}$. In this case k determines the product of the two fetching factors: $F_{n_1} \cdot F_{n_2} = \lceil \frac{k}{\Xi(\mathcal{G}) \cdot cs_{s_1} \cdot cs_{s_2}} \rceil = K'$, where F_{n_1} and F_{n_2} are positive integers. The number of $\langle F_{n_1}, F_{n_2} \rangle$ pairs can at most be $2 \cdot \lceil K' \rceil$, and we need to find the pair that minimizes the plan's cost. If the metric in use is, say, the sum cost metric, and the cost associated to a fetch of service s_i is c_i , the optimal values for the fetching factors are found by minimizing the expression $F_{n_1} \cdot t_{n_1}^{in} \cdot c_1 + F_{n_2} \cdot t_{n_2}^{in} \cdot c_2$. If n_1 and n_2 are not on the same path, $t_{n_1}^{in}$ and $t_{n_2}^{in}$ do not depend on F_{n_1} or F_{n_2} , so their optimal values are given by

$$F_{n_1} = \left\lceil \sqrt{K' \cdot \frac{t_{n_2}^{in} \cdot c_2}{t_{n_1}^{in} \cdot c_1}} \right\rceil \quad F_{n_2} = \left\lceil \sqrt{K' \cdot \frac{t_{n_1}^{in} \cdot c_1}{t_{n_2}^{in} \cdot c_2}} \right\rceil \quad (6)$$

If n_2 follows n_1 on the same path and uses as input some output from n_1 , then $t_{n_2}^{in}$ grows linearly with F_{n_1} , so the

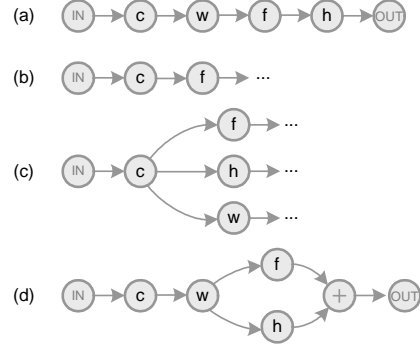


Figure 7: Running example: some alternative plans

optimal values are

$$F_{n_1} = 1 \quad F_{n_2} = \lceil K' \rceil \quad (7)$$

Under the same assumptions, we can generalize Equations 6 and 7, to the presence of n chunked services by respectively extracting the n -th root of an expression with more elaborated constants and by augmenting as much as possible the fetching factors for the last services in the chain.

EXAMPLE 5.1. We consider again the query and services from Example 3.1 and the choice of $\bar{\alpha}_1$ from Example 4.1, and optimize it according to the execution time metric. The only directly callable atom is *conf*, so every plan associated with the patterns $\bar{\alpha}_1$ will necessarily start by invoking this atom. Since all other atoms are callable after it, there are 19 alternative plans, obtained by using 6 different permutations and 13 parallelization options of the three other atoms.

Figure 7(a) sketches the DAG corresponding to the serial invocation of atoms in order of increasing *erspi* (i.e., *weather*, *flight*, and *hotel*), which is the plan suggested by the selective heuristics. This plan contains a single path, with $t_{conf}^{in} = 1$. The *weather* service is invoked with $t_{weather}^{in} = \xi_{conf}$ tuples, and produces $t_{weather}^{out} = \xi_{conf} \cdot \xi_{weather}$ tuples. Services *conf* and *flight* share input variables with their predecessor services, hence by using Equation 2 we obtain $t_{flight}^{in} = \min(\xi_{conf}, \xi_{conf} \cdot \xi_{weather}) = \xi_{conf} \cdot \xi_{weather}$ and $t_{hotel}^{in} = \min(\xi_{conf}, \xi_{conf} \cdot \xi_{weather}, \xi_{conf} \cdot \xi_{weather} \cdot \xi_{flight}) = \xi_{conf} \cdot \xi_{weather}$.

Assuming the *hotel* atom as the bottleneck node, we obtain:

$$\begin{aligned} ETM_1 &= F_{hotel} \cdot t_{hotel}^{in} \cdot \tau_{hotel} + \tau_{conf} + \tau_{flight} + \tau_{weather} \\ &= F_{hotel} \cdot \xi_{conf} \cdot \xi_{weather} \cdot \tau_{hotel} + \tau_{conf} + \tau_{flight} + \tau_{weather} \end{aligned}$$

where F_{hotel} depends on k . By computing ETM_1 , we obtain an upper bound and can prune other choices. For example, Figure 7(b) shows a partially constructed plan in which *flight* is chosen as the second service to be invoked; the estimated cost of this partial plan is $ETM_2 = t_{flight}^{in} \cdot \tau_{flight} + \tau_{conf}$. If $ETM_2 > ETM_1$, then we prune any plan having the nodes in Figure 7(b) as prefix. Under the same assumption, the choice of parallelizing *weather*, *flight*, and *hotel*, shown in Figure 7(c), is also pruned, since also its cost is greater than ETM_2 .

A plan that parallelizes *flight* and *hotel* after *weather* is shown in Figure 7(d). Since no decay is known for either *hotel* or *flight*, merge-scan is used; the join's estimated *erspi* is 0.01. The plan corresponding to these choices is shown

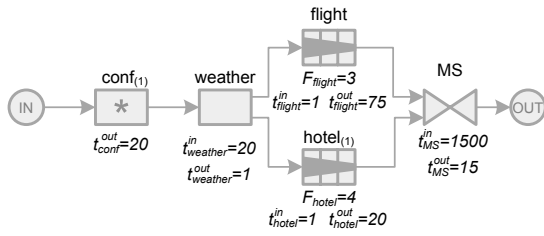


Figure 8: Running example: a physical access plan fully instantiating the plan of Figure 6

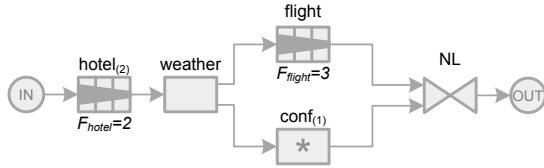


Figure 9: Running example: another access plan

in Figure 8, annotated with the actual values required for producing $k = 10$ result tuples. Some of the profiling parameters used for services are shown in Table 1. The calculation of the fetching factors for this plan is done by applying Equation 6, since `hotel` and `flight` are in parallel. This plan turns out to be optimal for the given choice of parameters according to the execution time metric.

Figure 9 shows another plan corresponding to different choices (access patterns, topology, fetching factors) for the same query, without reporting the t^{in} and t^{out} factors. ■

6. EXPERIMENTS

In order to test the assumptions made about services throughout this paper and to assess the quality of query outputs, we have implemented a set of services that allow us to access deep Web data. Specifically, we have implemented the `conf`, `weather`, `flight`, and `hotel` services adopted in our running example, by using available services or wrapping data sources on the Web. For instance, the implementation of the `flight` service is based on data coming from www.expedia.com; the site implements an interactive service accepting user-oriented data input and a confirmation procedure, and producing chunks of XML information that includes the tuples as modeled in `flight`; therefore we had to implement our own version of the service, that transparently performs the protocol and produces clean output tuples. Similarly, we built the other services from public-domain services: `conf` takes data from www.conference-service.com, `weather` from www.accuweather.com, and `hotel` from www.bookings.com. Our services are coded in Java (based on regular expressions) and executed locally on our test server, which supports a rudimentary query engine capable of performing both sequential and parallel joins; rewritten services feature chunking as in the respective source; this allows us to fetch a new chunk of results with Web service calls, so that the execution model of experiments is consistent with the execution model assumed by our this paper.

Table 1 reports the profiles of the four services with the data that are necessary for query optimization. Profiling information is derived from several test queries that have been individually issued to the different services.

Service	Type	Chunk size	Average response size	Average response time
conf	exact	-	20	1.2
weather	exact	-	0.05	1.5
flight	search	25	-	9.7
hotel	search	5	-	4.9

Table 1: Characterization of the example services

Figure 10: Screenshot of results for the optimal plan

The most interesting comparative experiments are concerned with different query plans for the same query or different levels of caching for the same plan, as discussed next.

The first experiment measures the execution time and the number of service calls for query plans in the no-cache setting; we have chosen the optimal query plan obtained analytically (see Example 5.1), and the two query plans suggested by the serial and parallel heuristics; the three plans are shown in Figure 7, and we will indicate plan (a) as S , plan (c) as P , and plan (d) as O . The results are shown in the three leftmost bars of the chart of Figure 11. All the plans make exactly one call to `conf`, which returns 71 tuples, corresponding to 54 different cities (some cities host several events). Plan S filters the 71 tuples through `weather`, and only 16 cities remain. The results proliferate through `flight`, and then traverse the `hotel` node. Plan P immediately invokes `weather`, `flight`, and `hotel` in parallel. This turns out to be the worst choice, since the selective effect of `weather` is lost, which weighs the heaviest on the slowest service (`flight`). The optimal plan O calls `hotel` and `flight` in parallel after `weather`. The improvement between S and O is significant: redundant calls (72%) on `hotel` are removed by construction of the plan; however, the overall time improvement is not as high, since the saved calls are cached on the server of Bookings.com and are therefore answered very quickly. The experiment confirms that the predicted optimal plan is indeed better than the other choices, and both the time and the service calls saved by O are significant.

Figure 10 shows a screenshot of the answer produced by the execution engine on the running example query, in which the constants concerning the conference topic and the temperature have been asked as parameters; the figure reports the results obtained with plan O on a no-cache setting.

The second experiment measures the effect of different cache settings on query plan execution. We have tested O , S , and P in the three different cache settings considered in this paper. Along with the already discussed results for the no-cache setting, the corresponding values for the one-

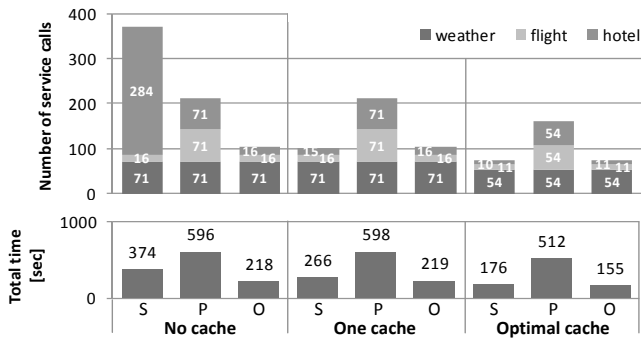


Figure 11: Measures for the tested plans (calls per service, and total times in seconds).

call cache and optimal cache settings are also reported in Figure 11. In the no-cache setting, the results proliferate through *flight*, and then traverse the *hotel* node. When executing *S* in the one-call cache setting, the redundancy provoked by *flight* is taken care of, and no more than 16 calls will be forwarded to *hotel* (in fact, only 15 will flow, since for one city no flight is found). The optimal cache allows reducing the number of calls already to *weather* by removing duplicate cities (54 distinct cities out of 71); the overall number of calls is greatly reduced with the optimal cache, since duplicates are disregarded from the start. Note that no improvement can be observed for *O* (and, similarly, for *P*) between the execution in the no-cache and in the one-call cache setting, since no further redundant calls are captured by the cache. Instead, the improvement is apparent with the optimal cache because a few duplicate calls are avoided (and Expedia does not cache such calls). The experiment confirms that the benefits of caching are significant, and may be remarkable already in the one-call cache setting, although this highly depends on the structure of the plan.

Further improvements are obtained by leveraging parallelism on the server side, when services are known to support it. We have done this as a separate test, where we have immediately dispatched all the available calls to a service on parallel threads. For example, in *S*, all the 71 calls to *weather* can be dispatched to parallel threads, and so can those to *flight* (and, later, to *hotel*); the overall execution time, measured in 76s, will then amount to the sum of the heaviest calls to *weather*, *flight*, and *hotel*, plus some overhead due to multi-thread management. However, the performance of the one-step cache will be affected by such a multi-threading, since the order in which duplicate results will appear is randomized by parallelism (the 284 calls to *hotel* have reduced to 212 with the one-call cache setting in our experiments with multi-threading, whereas they reduced to 16 without it). Of course, the optimal cache suffers no such drawbacks.

We have also applied the framework to other domains, such as news management, bibliographic search, and bioinformatics. For example, we were able to query protein repositories to find evolutionary relationships between human and mouse proteins including repeated protein domains and involved in the glycolysis metabolic pathway, using the InterProt (<http://www.ebi.ac.uk/interpro/>), UniProt (<http://www.uniprot.org/>), BLAST (<http://www.ebi.ac.uk/blast2/>), and KEGG (<http://www.genome.jp/kegg/>) data sources.

7. RELATED WORK

The most closely related work is [16], in which the authors propose a Web service management system (WSMS) that enables querying multiple Web services in a transparent and integrated fashion, similarly to the problem approached in this paper. The authors propose an algorithm for arranging a query's Web service calls into a pipelined execution plan that exploits parallelism among Web services to minimize the query's total running time under the bottleneck cost metric. They assume all services to be exact and with no chunking of results, and model them by means of their per-tuple response time and selectivity; in the queries they consider, all input attributes get their values from either exactly one other Web service or from the user's input.

Answering queries over Web services. Historically, answering queries over independent data sources has been the research object of *parallel* or *distributed query processing* [10, 17, 7]. Two main techniques have emerged in this research field: code shipping and data shipping. The latter technique is heavily leveraged in our work, and data are shipped in a pipelined fashion from one service to another, so as to maximize parallelism.

The coordinated execution of distributed Web services is the subject of *Web services composition*, which comes in two different flavors: orchestration and choreography. The distributed approach of *choreographed* services (e.g., using WS-CDL [20] or WSCI [19]) does not suit our query processing problem, because choreographies are not executable and require the awareness of and compliance with the choreography by all the involved services. The centralized approach of *orchestrated* services (e.g., using BPEL [13]) suits better the research problem addressed in this paper, as orchestrations are executable service compositions (i.e., query plans, in our terminology) and services need not be aware of being the object of query optimization and execution. In the specific case of BPEL, however, its workflow-based approach does not provide the necessary flexibility when the invocation order of services needs to be computed at runtime, as is our case (e.g., dynamically fixing a number of fetches to be issued to a service remains hard).

Finally, Yahoo Pipes⁵ and IBM DAMIA⁶ [1] enable a Web 2.0 approach to compose ("mash up") queries over distributed data sources like RSS/Atom feeds, comma-separated values, XML files, and similar. Both approaches come with user-friendly Web interfaces, which allow users to draw workflow-like data feed logics based on nodes representing data sources, data transformations, operations, or calls to external Web services. Unlike the techniques discussed in this paper, both Pipes and DAMIA require the user to explicitly specify the query processing logic procedurally, which is generally not a trivial task for unskilled users; we automatically derive a plan from a declarative query formulation.

It is worth noting that the previous service querying approaches effectively enable users to distribute a query over multiple Web services, but they do not focus on the peculiarities of search services, such as ranking and chunking.

Answering queries under access limitations. The issue of processing queries under access limitations, by some authors studied under the headline of *binding patterns*, has been widely investigated in the literature [14, 11, 12, 21, 6].

⁵<http://pipes.yahoo.com/pipes/>

⁶<http://services.alphaworks.ibm.com/damia/>

In this paper, we have assumed that queries are always designed so as to admit at least one permissible choice of access patterns. However, for some queries, it may happen that no permissible choice of access patterns exists. Although, in this case, the original user query cannot be answered, 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. In particular, such “off-query” services may be invoked so that their output fields provide useful bindings for the input fields of the services in the query with the same abstract domain. In our running example, one could conclude, e.g., that `conf.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 used to find some of the answers to the query. This query expansion can only provide an approximation of the original query that, in general, requires the evaluation of a recursive query plan even if the initial query was non-recursive [12].

Also, since accessing data sources over the Web is typically a costly task, later works have addressed the issue of reducing the accesses to the sources, while still returning all obtainable answers. For instance, some optimizations to be made during query plan generation to minimize the accesses to data sources are discussed in [11] for a subset of conjunctive queries, named *connection queries*; more expressive classes of queries, including conjunctive queries, are covered in [5].

Optimizing query plans. We propose a branch-and-bound query construction method driven by a set of heuristics that allow us to take into account the peculiarities of search services and to converge to an optimal solution with respect to a given cost metric. Other well-known query optimization techniques exist, such as *transformation*-based approaches [15] or *randomized* approaches [9]. Like in [8], in our case we cannot easily resort to transformation-based techniques, as in general access patterns and ranking orders do not allow us to guarantee properties like associativity and commutativity for joins. The use of randomized approaches, on the other hand, is not efficiently practicable in presence of explicit access patterns, as such would typically lead to uselessly consider a large number of infeasible plans during query optimization.

Branch-and-bound algorithms are, e.g., adopted in [18], which considers a query optimization problem with characteristics that are similar to our problem. Specifically, the authors focus on ranked queries in the context of classical databases, where the “ranking” is expressed by means of an *explicit* preference function over the values of a tuple’s attributes, to be taken into account when computing top-k answers. However, service characteristics and implicit ranking orders are not dealt with, which instead are a distinguishing feature of the work presented in this paper.

8. CONCLUSIONS

This paper has presented an overall framework for expressing and optimizing multi-domain queries on the Web. The paper highlights the relevance of search services and rankings in such queries, and adapts to this new framework classical models and methods successfully used by relational query optimizers.

Acknowledgements All authors acknowledge support from Italian project “New technologies and tools for the integration of Web search services”, PRIN Call 2007-08.

9. REFERENCES

- [1] M. Altinel, P. Brown, S. Cline, R. Kartha, E. Louie, V. Markl, L. Mau, Y.-H. Ng, D. Simmen, and A. Singh. Damia: a data mashup fabric for intranet applications. In *VLDB’07*, pages 1370–1373. VLDB Endowment, 2007.
- [2] M. K. Bergman. The deep web: Surfacing hidden value. *The Journal of Electronic Publishing*, 7(1), 2001.
- [3] D. Braga, D. Calvanese, A. Campi, S. Ceri, F. Daniel, D. Martinenghi, P. Merialdo, and R. Torlone. Ngs: a framework for multi-domain query answering. In *Proc. of IIMAS’08, ICDE 2008 workshop*, pages 254–261, 2008.
- [4] D. Braga, A. Campi, S. Ceri, and A. Raffio. Joining the results of heterogeneous search engines. *Information Systems*, 2008. To appear.
- [5] A. Cali and D. Martinenghi. Querying data under access limitations. In *Proc. of ICDE 2008*, pages 50–59, 2008.
- [6] A. Deutsch, B. Ludäscher, and A. Nash. Rewriting queries using views with access patterns under integrity constraints. *Theoretical Computer Science*, 371(3):200–226, 2007.
- [7] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, 1990.
- [8] D. Florescu, A. Y. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *SIGMOD’99*, pages 311–322, 1999.
- [9] Y. E. Ioannidis and Y. Kang. Randomized algorithms for optimizing large join queries. *SIGMOD Rec.*, 19(2):312–321, 1990.
- [10] Z. G. Ives, A. Y. Halevy, and D. S. Weld. Adapting to source properties in processing data integration queries. In *SIGMOD’04*, pages 395–406, 2004.
- [11] C. Li and E. Chang. Answering queries with useful bindings. *ACM Trans. Database Syst.*, 26(3):313–343, 2001.
- [12] T. D. Millstein, A. Y. Levy, and M. Friedman. Query containment for data integration systems. In *PODS’00*, pages 67–75, 2000.
- [13] OASIS. Web Services Business Process Execution Language. Technical report, <http://www.oasis-open.org/committees/wsbpel/>, 2007.
- [14] A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns. In *PODS’95*, pages 105–112, 1995.
- [15] P. Seshadri, J. M. Hellerstein, H. Pirahesh, T. Y. C. Leung, R. Ramakrishnan, D. Srivastava, P. J. Stuckey, and S. Sudarshan. Cost-based optimization for magic: algebra and implementation. *SIGMOD Rec.*, 25(2):435–446, 1996.
- [16] U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query optimization over web services. In *VLDB’06*, pages 355–366. VLDB Endowment, 2006.
- [17] M. Tamer Ozsu and P. Valduriez. *Principles of distributed database systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.
- [18] Y. Tao, V. Hristidis, D. Papadias, and Y. Papakonstantinou. Branch-and-bound processing of ranked queries. *Inf. Syst.*, 32(3):424–445, 2007.
- [19] W3C. Web Service Choreography Interface (WSCI) 1.0. W3C Note, August 2002.
- [20] W3C. Web Services Choreography Description Language Version 1.0. W3C Working Draft, December 2004.
- [21] G. Yang, M. Kifer, and V. K. Chaudhri. Efficiently ordering subgoals with access constraints. In *PODS’06*, pages 183–192, 2006.