Zuerst ersch. in : Search Computing : Challenges and Directions / Marco Brambilla; Stefano Ceri. - Springer : Berlin, 2011. - S. 188-210. - (Lecture Notes in Computer Science ; 5950). - ISBN 978-3-642-12309-2
http://dx.doi.org/10.1007/978-3-642-12310-8_10

Chapter 10: Join Methods and Query Optimization

Daniele Braga, Stefano Ceri, and Michael Grossniklaus

Dipartimento di Elettronica e Informazione, Politecnico di Milano,
Piazza Leonardo da Vinci 32, 20133 Milano, Italy
{braga,ceri,grossniklaus}@elet.polimi.it

Abstract. Joins between data sources are an essential ingredient of multi-domain queries, as they exploit connection patterns defined between service marts or between service interfaces. This chapter moves from the definition of a query language over service interfaces, sketching how queries can be directly expressed over service marts and how these can be translated over service interfaces. The fundamental operation discussed in this chapter is the binary join between two sources, which is influenced by the type (search vs. exact) of services and by the management (parallel vs. sequential) of service calls. Then, this chapter presents an optimization framework for queries over several service interfaces, which considers several cost metrics for mapping queries into query plans, consisting of specific operations over services, and includes a branch and bound approach to the exploration of the combinatorial search space of all possible query plans.

1 Introduction

This chapter delves into the issues of formulating and optimizing multi-domain queries over several services, focusing on the specific problems that arise due to the presence of search services in the queries. 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 pages of results. Thus, a query strategy that retrieves all the answers from a search service is rarely 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 answers in ranking order. Thus, while composing results from multiple services, answers should be presented according to a global ranking that is obtained either as an exact composition function of the rankings (then, we can talk about "top-k" results) or an approximation of that function.

In this chapter, we define a **formal model** for the optimization and the execution of multi-domain queries over services which expose heterogeneous information sources. Such model serves as a unifying perspective for several diverse possible application settings, ranging from providing expert users with service mash-up tools to providing the foundations for vertical service integration frameworks. The originality of the model stems from the way in which data sources are classified, distinguishing

between *exact* services, that have a "relational" behavior and return either a single answer or a set of unranked answers, and *search* services, that return a list of answers in ranking order, according to some measure of relevance.

We also formally define **query plans**, playing the same role within our context as physical access plans in relational databases. A plan is defined as the orchestration of service invocations, possibly in parallel, which takes into account some significant features of the service, including its ability to return results in chunks. Query plans schedule the invocations of Web services and the composition of their inputs and outputs. Within plans, the main operations are **joins** between results, whose execution can take place according to several join strategies.

Then, we define an **optimization method** for choosing the "best" access plan, i.e. the one that optimizes an objective function, resorting to a classical and well-grounded approach such as branch and bound in order to efficiently explore the solution space. Cost minimization is performed against one of several alternative *cost metrics*, capturing different scenarios and different optimization goals.

The organization of this chapter is as follows. Section 2 presents the state-of-the-art of query optimization, with a specific emphasis upon systems that integrate data collected through Web services. Section 3 presents the query language, offering its syntax, semantics, and exemplification on the running example. It then presents a graph model for query plans in which nodes denote operations and arcs denote the dataflow between them. The most important operation is the join of two services, and Section 4 focuses on join methods. Finally, Section 5 shows the optimization steps and heuristics leading to the selection of an optimal plan.

2 State of the Art

The background for this chapter ranges over several disciplines. In this Section we move from the classical foundations of query processing and optimization, in order of increasing specificity, to Web service composition, query answering under access limitation, and the study of systems dedicated to query optimization over Web services (Web Service Management Systems).

2.1 Query Processing Foundations

Query processing is perhaps the fundamental technology offered by database management systems, in the sense that they provide means for translating declarative SQL queries into highly efficient access plans. The main merit of query processing is the ability of producing high performance plans. Work in query processing can be traced back to foundational papers such as [7]. Dedicated books, such as [30], focus just on query processing. In most approaches, query processing consists first in giving an abstract representation of the query (typically in the form of a directed acyclic graph having data resources as leaves and data extraction and transformation operations as intermediate nodes [7], and then using equivalence transformations upon such graphs in order to use the best possible operations and settings of operations' parameters, so as to minimize an objective function. Such processing is called query optimization and typically uses methods of operations research

(e.g., [15]). Extensible query processors are obtained by describing equivalence transformations as rules, and then varying the rule set by adding or dropping rules or by varying their priority [17]. Other well-known query optimization techniques exist, such as *transformation*-based approaches [21] or *randomized* approaches [13].

Branch-and-bound algorithms are, e.g., adopted in [24], 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 chapter. 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. Like in [12], in our case we cannot easily resort to transformation-based techniques, as the presence of access patterns and ranking orders does not guarantee properties like associativity and commutativity for joins. Randomized approaches, on the other hand, are not efficiently applicable with explicit access patterns, as this would typically lead to uselessly consider a large number of infeasible plans during query optimization.

2.2 Answering Queries over Web Services

Historically, answering queries over independent data sources has been the research object of *parallel* or *distributed query processing* [14][23][11]. Two main techniques have emerged in this research field: code shipping and data shipping. While code shipping to Web services is not feasible, data shipping is feasible and allows the feeding of results coming from one service in the access plan to another service in the plan. The latter technique is heavily leveraged in our work, as data are shipped in pipelines 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 [9]. The distributed approach of *choreographed* services (e.g., using WS-CDL [26] or WSCI [27]) 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 [19]) 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). There is a growing amount of semantic approaches to the runtime composition of Web services (e.g., [28] or [8]), but their focus is typically on functional requirements or quality of service [2] and less on data.

Inspired by the work presented in [22], Tatemura et al. [25] introduce the idea of continuous query over service-provided data feeds (e.g., through RSS or Atom). The

goal is to mash up and monitor the evolution of third-party feeds and to query the obtained result. Their mash-up query model is articulated into collections of data items and collection-based streams of data (streams also track the temporal aspect of collections and allow the querying of the history of collections). Suitable select, join, map, and sort operators are provided for the two constructs. The described system consists of a visual mash-up composer, an execution engine, and interfaces for users to subscribe to mash-up feeds equipped with personalized queries.

Finally, Yahoo Pipes[1] and IBM Damia[2] [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 and intuitive 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. 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, especially for the case of joins, which have to be explicitly programmed by the user. Instead, with our approach 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 specifically focus on the peculiarities of search services, such as ranking and chunking. Characteristics like the ranking order of results or advanced querying techniques are not considered.

2.3 Answering Queries under Access Limitations

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 by characterizing service parameters as binding patterns, i.e., classifying them as input or output parameters, thereby clarifying the different ways in which it can be invoked. 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 [20][16][18][29][10].

In our work, we have assumed that queries are always designed so as to admit at least one 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. A query "augmentation" of this kind, however, can only provide an approximation of the original query that, in

[1] `http://pipes.yahoo.com/pipes/`
[2] `http://services.alphaworks.ibm.com/damia/`

general, requires the evaluation of a recursive query plan even if the initial query was non-recursive.

The problem of finding all obtainable answers to a query posed over data sources with access limitations has been studied in [18] and later works. It has been shown that, even though a query is conjunctive, finding all obtainable answers in general requires a recursive query plan. 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 [16] for a subset of conjunctive queries, named *connection queries*. More expressive classes of queries, including conjunctive queries, are covered in [6].

2.4 Web Service Management Systems

The paper in which Srivastava et al. [22] introduced the notion of Web Service Management System can be considered as one of the main inspiration sources of the query optimization framework in SeCo. 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 a bottleneck cost metric (i.e., by choosing to optimize the execution of the slowest service). 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. They focus upon simple queries where all input attributes get their values from either exactly one other Web service or from the user's input, and did not consider the peculiarity of search services, with ranked results.

Inspired by [22], Braga et al. developed in [4] the foundations of the optimization framework for Search Computing that is here addressed in Section 5.

3 Query Formulation and Translation into an Executable Plan

The optimization of a query over a set of service marts starts from a formulation of the query in a conjunctive query language and ends with a fully instantiated invocation schedule. The execution environment to which the invocation schedule is addressed is a system capable of executing query plans (as they will be formally defined in the sequel). This means that the system can execute requests, collect their results, and integrate them progressively, forming the answers as combinations of partial invocation results.

3.1 Query Formulation

We consider select-join queries on service marts and connection patterns. Our formalism abstracts away from the details of any underlying representation, and for each information source resorts to simple service definitions as illustrated in Chapter 9. No projection is performed, as all the data of a service implementation are

presented to the liquid query interface, where the format of results includes projecting over some of the service mart attributes. The user interface may also present one copy of all attributes which are set equal by a query. Queries can be expressed, with exactly the same syntax and semantics, either over service marts or over service interfaces. In the former case, the query processor must select suitable service interfaces so as to make the query *feasible*, according to the definition given below. In this chapter (and in our first conceptualization of Search Computing) we assume that users operate directly over service interfaces. Hence, the interface selection process, though mentioned later on with respect to query optimization, is not part of the query processing chain.

More formally, a **query** consists of a set of services $s_1, \ldots, s_n$ (the same service can occur several times with a different renaming for each different use), a set of selection predicates, and a set of join predicates.

We recall from the previous chapter that an attribute of a service can be either an atomic attribute or a repeating group. A repeating group consists of a non-empty set of atomic sub-attributes that collectively define one property of an object. Atomic attributes are single-valued, while repeating groups are multi-valued. We indicate an attribute A of a service s as $s.A$. A sub-attribute A of a repeating group $s.R$ is indicated as $s.R.A$. If no ambiguity arises, the prefixes s or $s.R$ may be omitted.

A selection predicate is an expression of the form $A\ op\ const$, where A is an atomic attribute or sub-attribute, *const* is a type-compatible constant, and *op* is a comparator among {=, <, <=, >, >=, like}. A join predicate is an expression of the form $A\ op\ B$, where A and B are type-compatible attributes or sub-attributes, and *op* is a comparator among {=, <, <=, >, >=, like}.

A service s from a query is **reachable** if, for every input (sub-)attribute A of s, the query contains a selection predicate of the form $A=const$, or a join predicate of the form $A = B$ where B is a (sub-)attribute of a reachable service. A query is **feasible** if all its services are reachable.

A tuple of a service is a mapping that sends each attribute $s.A$ into a value of the domain of A. For a tuple t of s, we use the notation $t.A$ to indicate the value of t for attribute $s.A$. Note that, if $s.R$ is a repeating group, the value $t.R$ is a set of tuples over the sub-attributes of $s.R$.

The **semantics** of a feasible query over $s_1, \ldots, s_n$ with a set P of (selection and join) predicates is defined as the largest set of composite tuples of the form $t_1 \cdot \ldots \cdot t_n$ such that the following two conditions hold:

1. $t_i \in s_i$ for $1 \leq i \leq n$; and
2. there is a mapping M from each repeating group of the form $s_i.R$ occurring in P into a tuple $M(s_i.R)$ in $t_i.R$ such that each expression in the set obtained from P by

    - replacing each occurrence of $s_i.R$ with $M(s_i.R)$ and, after that,
    - replacing each occurrence of s_i with t_i

    is satisfied according to the natural interpretation of comparators.

Consider two services S_1 and S_2 over the repeating group R with sub-attributes A and B. Assume that S_1 provides two objects t_1=({<1,x>,<2,x>}), t_2=({<2,x>,<1,y>})

and that S_2 provides two objects t_3=({<1,x>,<2,y>}), t_4=({<2,x>}). Thanks to the above choice of semantics, the query *Q_1: select S_1 where S_1.R.A=1 and S_1.R.B=x* produces the result {t_1}, and the query *Q_2: select S_1, S_2 where S_1.R.A=S_2.R.A and S_1.R.B=S_2.R.B* produces the result {$t_1 \cdot t_3$, $t_1 \cdot t_4$, $t_2 \cdot t_4$}. Note that t_1 belongs to Q_1's result because S_1.R can be replaced by the tuple <1,x> of t_1.R and the resulting expressions 1=1 and x=x are trivially satisfied. Informally, t_1 is selected because one of its repeating groups satisfies the selection condition. Conversely, t_2 does not belong to Q_1's result because, although its sub-attributes separately satisfy the selection, this occurs in different tuples of the repeating group. Therefore no individual tuple of the repeating group satisfies the selection condition. Similarly, note that the tuple $t_2 \cdot t_3$ does not belong to Q_2's result because, although its sub-attributes satisfy the join condition, this occurs in different tuples of the repeating group.

Instead of constants in the query we may also use variables prefixed as INPUT, whose value is provided by users at query execution time. Using the above syntax and semantics, the example query can be expressed as follows:

RunningExample:

```
Select Movie11 As M, Theatre11 as T, Restaurant11 as R
where
(selection conditions)
      M.Genres.Genre=INPUT1 and M.Openings.Country=INPUT2 and
      M.Openings.Date>INPUT3 and T.UAddress=INPUT4 and T.UCity=INPUT5
      and T.TCountry=INPUT2 and T.Category.Name=INPUT6 and
(join conditions)
      M.Title=T.Title and T.TAddress=R.RAddress and T.TCity=R.RCity
      and T.TCountry=R.RCountry.
```

Note that the condition *M.Openings.Country=INPUT2 and M.Openings.Date>INPUT3* extracts movies such that a single opening tuple satisfies both the conditions on country and date.

Join predicates used by a query are normally establishing join condition over connection patterns. Therefore, join conditions can be expressed in a more compact way by mentioning connection patterns, yielding to the formulation below:

RunningExample:

```
Select Movie11 As M, Theatre11 as T, Restaurant11 as R
where Shows(M,T) and DinnerPlace(T,R) and
    M.Genres.Genre=INPUT1 and M.Openings.Country=INPUT2 and
    M.Openings.Date>INPUT3 and T.UAddress=INPUT4 and T.UCity=INPUT5
    and T.TCountry=INPUT2 and T.Category.Name=INPUT6
```

Checking the feasibility of this query is immediate by considering that all input places of $Movie_{11}$ and $Restaurant_{11}$ are associated with INPUT variables, hence they are reachable, and that by virtue of the join variables linking Theatre to Restaurant also Restaurant is reachable. If all services are properly designed and registered, queries over service interfaces whose join conditions include the connection patterns and whose selection conditions include an equality predicate with either a constant or an input variable are feasible queries. Tools for drawing queries upon the graph representation of service marts and connection patterns can help query designers, so as to enable the drawing only of feasible queries [5].

We finally turn to expressing rankings, an important aspect of Search Computing queries. We assume that each service interface s_i is associated with a **scoring function** SF_i. If s_i is ranked, SF_i indicates how to obtain a score in the [0,1] interval as a function of the attributes of s_i[3]; if it is unranked, the SF_i is a fixed constant. Then, the query is associated with a **ranking function** f expressed as a sequence $(w_1, \ldots, w_n)$ of non-negative weights for the scores used in the query. If tuples $t_1, \ldots, t_n$ from, respectively, $s_1, \ldots, s_n$ are used to form a combination, the ranking function of the formed combination $t_1 \cdot \ldots \cdot t_n$ is given as $w_1S_1 + \ldots + w_nS_n$, where S_i is the score of t_i for $1 \leq i \leq n$; the weight of unranked services is set equal to 0. In the above query, with 3 ranked service interfaces, a possible ranking function is (0.3, 0.5, 0.2).

Ranking functions may be assigned prior to query execution, either at query definition time or at query presentation time. They can also be altered dynamically through the query interface, yielding to changes in the query execution strategy. Only ranking functions defined at query definition time can be used for query optimization.

3.2 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 production of **k tuples**, where k is a parameter of the optimization. Every tuple includes contributions from the various service calls which are progressively composed according to the dataflow. Result tuples can be guaranteed to be the **top-k** tuples according to the ranking function, or instead be just **k good** tuples, emitted with an approximation of the total order expressed by the ranking function. Top-k tuples are generated by query plans which use top-k join methods, described in the next chapter. All other plans use the join methods described in Section 4, which do not guarantee top-k results, but are normally faster than top-k join methods.

We represent plans as directed acyclic graphs (DAGs) where:

- Every node represents either an atom in the conjunctive query (i.e., a service invocation), or a join, or a selection operation.
- Every arc indicates data flow and parameter passing from outputs of one service to inputs of another service.
- Atoms are partitioned into *exact* and *search* services. Exact services are distinguished between *proliferative* and *selective* and may be chunked, while search services are always proliferative and chunked. An exact service is selective if it produces in average less than one tuple per invocation (and therefore, in average, fewer output tuples than input tuples). An exact service that is not selective "per se" is said to be selective *in the context of a query* when the query includes a selection predicate over the output attributes of the service and the combined execution of the exact service call and of the selection produces fewer output tuples than input tuples.
- Joins are either performed as *pipe joins* or as *parallel joins*. Pipe joins occur when the query in the plan is made feasible through a strategy which induces an

[3] The case of opaque rankings can be dealt with by associating the position of tuples in the result with a new attribute and then translating the position into a score in the [0..1] interval.

I/O dependency between two services, whereas parallel joins occur when there is no such dependency. Parallel joins are represented by explicit nodes, marked with an indication of the join strategy to be employed, while pipe joins are represented simply by cascading two service invocations.

- Selection nodes express selection or join predicates which cannot be performed either by calling services or by using connection patterns.[4] Each predicate is independently evaluated on tuples representing intermediate or final query results, immediately after the service call that makes the selection or join predicates evaluable.
- Two explicit nodes represent the query input (i.e., the process of reading INPUT variables, mapping onto the arguments of services and joins, and starting query execution) and output (i.e., returning tuples to the query interface).

The graphical syntax for representing query plans is represented in Fig. 1.

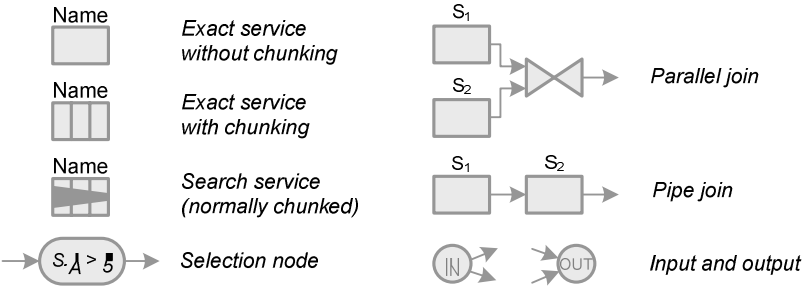

Fig. 1. Elements of query plans

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 selectivity and sizes of results returned by each service call. Cost models use estimates of the average result size of exact services and of chunk sizes.

An example of query plan representing the access to four services is shown in Fig. 2. The plan consists first in accessing two exact services named Conference and Weather. Conference is proliferative and produces 20 conferences on average, while Weather is selective in the context of the query, because extracted tuples are checked against the condition that the average temperature at the time of the conference must be above 26°C, and thus many of them can be discarded. Then, services describing flights to the conference city and hotels within that city are called, and their results are joined according to a given strategy, called merge-scan (MS), to be discussed later. Results of the join are transmitted to the user interface by the output node.

[4] These are expressed in the query and have the form: $S_i.att_i$ *op const* or $S_i.att_i$ *op* $S_j.att_j$ where *op* is any comparison operation and attributes can be either single or multi-valued.

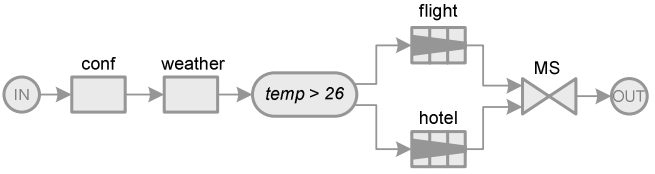

Fig. 2. Example of query plan

In our framework, we retrieve only the fraction of tuples of proliferative search services that are sufficient to obtain the first k tuples as query answers. We set k as optimization parameter so that the answered tuples should normally satisfy the user's needs (e.g. k=10), but a plan execution can be continued, after an explicit user request, thereby producing more tuples. Therefore, 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 input keywords and resubmit the same query, or turn to a different query or Web activity. Moreover, if the strategy does not guarantee top-k results, a query interface can be set so as to retrieve continuously tuples from the execution engine, without waiting for the extraction of k tuples. More details about user interaction are delayed to Chapter 13.

Also, for each node N in the plan we shall estimate the number of tuples in output, denoted as t^{out}_N. We assume that the user always injects one single input tuple in the plan, represented by the start node. For exact services, t^{out}_N is given by the product of t^{in}_N (the number of input tuples) with the service's average cardinality. For selection nodes, t^{out}_N is equal to t^{in}_N multiplied with the selectivity of the predicate. In both cases, numbers descend from the static properties of the query and can be computed from service interface statistics, under suitable independence and value distribution assumptions. Instead, if node n represents a join, t^{out}_N depends on the join selectivity and on the join method used; and if a node represents a search service, t^{out}_N is given by the product of the chunk size with the total number F_S of fetches determined by the plan, which may in turn depend on the input t^{in}_N. Therefore, the main decisions to be taken are join methods and access to search services. An annotated plan whose nodes are associated with t^{in}_N, t^{out}_N, and F_S (if appropriate) is denoted as a fully instantiated query plans and can be associated with an execution cost. Fig. 3 shows an example of fully instantiated query plan obtained by annotating the plan of Fig. 2.

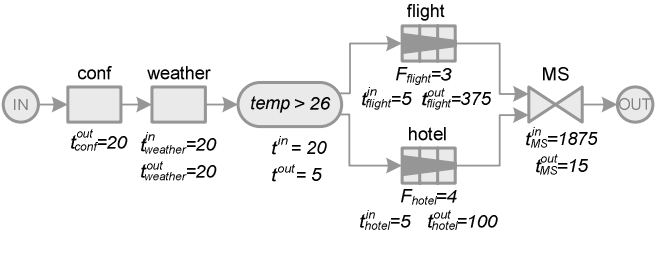

Fig. 3. Fully instantiated query plan, with annotations

4 Join Methods for Search Computing

In this section, we explore different join methods for search computing. We start by outlining the problem space and then continue to discuss three orthogonal characteristics of join methods, topology, invocation, and completion strategy. Finally, we discuss a number of concrete join methods that serve as a basis for upcoming chapters.

4.1 Problem Statement

Consider the join of two search services S_X and S_Y, and let R be the result of the join. R is a sequence of tuples r_k each obtained by joining two tuples x_i, produced by S_X, and y_j, produced by S_Y; r_k is associated with a ranking function ρ^R_k, producing values within the [0..1] interval, obtained as the weighted sum of two scoring function ρ^X_i computed over t_i and ρ^Y_j computed over t_j. We can represent the chunks 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 chunks decreases from the origin down to the end of the list (see Fig. 4). Each point P in the plan represents a couple (x_i, y_j) which must be joined. If the join predicate holds, the point P belongs to the result. Services S_X and S_Y produce at each call a new chunk, named c_{Xi} and c_{Yj} respectively, where c_{Xi} is a chunk returned by S_X in response to its i-th call and c_{Yj} is a chunk returned by S_Y in response to its j-th call. The Cartesian plan is thus divided into rectangles with $n_X \cdot n_Y$ points, where n_X and n_Y represent the chunk size of each service. We call ***tile*** t_{ij} the rectangular region that contains the points relative to chunks b_{Xi} and b_{Yj}. 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 a join operation. Each rectangular region of size m·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.

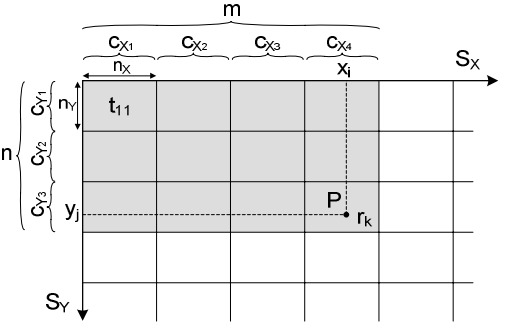

Fig. 4. Search Space for join operations

A join strategy is **optimal** if it produces, with the minimum cost, k tuples with the top-k highest ranking. The cost of a strategy depends on the adoption of a specific cost model, whose factors include the cost of interacting with services and the cost of computing the join between service results. Cost-based optimal strategies for joining search services are the focus of the next chapter.

However, top-k optimality is neither precise enough nor practically desired. First, rankings are sometimes approximate and the ranking function is rather arbitrary, thus reducing the practical relevance of producing top-k results. Second, it may be inappropriate to produce results in strictly decreasing ρ^R values, because achieving such result normally requires halting the output production of result tuples until the system decides that all top-k tuples are produced. So we introduce a revised notion of optimality, which only an approximate ranking of results.

If we assume that services return results in decreasing ranking order, we say that a join strategy is **extraction-optimal** if it produces elements r_k in decreasing order of the product of the two rankings $\rho^X \cdot \rho^Y$ and with the minimum cost. Such notion extends from tuples to tiles by using the ranking of the first tuple of the tile as representative for the entire tile. Extraction-optimality enables the presentation of results which satisfy the join condition in the order in which they are computed, tile by tile. Therefore, the dataflow of results produced to the user is not "blocking" (by abusing of the terminology which is typical of query streams), and results can be presented to users while they are extracted from the search engines, typically arranged in chunks. The notion of extraction optimality can be further refined to be interpreted in *global* sense, i.e. relative to all the tiles in the search space, or in *local* sense, i.e. relative to the tiles already loaded in the search space and available to the join operation. If two tiles are adjacent, then the one with smaller index sum is extracted first by extraction-optimal methods.

Concerning the cost model, we consider the scenario in which the cost of join execution is dominated by request-response execution. We assume that once a chunk is retrieved as the effect of a request-response to services, then join requires simple main-memory comparison operations and can be neglected[5]. We further characterize the way in which ranking decreases (from top values close to 1 down to bottom values close to 0), by subdividing search services in the following two classes:

1. **Search Services with Step Scoring Function.** We assume that, by performing a limited number h of request-responses, most of the relevant entries will be retrieved, because the entry scores decrease with a deep step after h request-responses. We assume h to be a parameter associated with the service.
2. **Search Services with Progressive Scoring Function.** We assume that the scoring function decreases progressively, with no step. This case accommodates all regularly decreasing functions, e.g. linear or square value distributions.

[5] In previous work [3] we considered also the scenario where the cost of request-response execution is dominated by join execution. Such scenario considers more expensive "join" operations, e.g. the matching of terms which are extracted from a taxonomy or an ontology, where matching is expensive, e.g. because each element comparison requires itself a call to a semantic Web service.

Note that the unavailability of the ranking function does not affect our basic assumption that the search services return results in ranking order, but simply captures the situation in which this function is opaque. However, if the function is opaque, then classifying services and determining h in the former case is more difficult.

4.2 Topology

The first characteristic of a join method is its topology. When joining two search services, there are basically two possible ways of invoking the services. Either the services are invoked *sequentially* or *in parallel*. In the context of this book, the former case is referred to as a **pipe join**, while we will call the latter a **parallel join**.

4.2.1 Pipe Joins

Pipe joins use the fact that the access patterns of certain search services accept input parameters. In a sequence of services, the first service returns chunks of tuples that are passed down the sequence. A subset of the attributes of these tuples is the set of join attributes of a pipe join, whose values are passed, or "piped", to another service that appears later in the sequence. These values are used as input values of the latter service's calls, so as to produce a set of result tuples, obtained by composing the input tuple with the service call results. Note that in order to perform a pipe join, the two search services do not need to follow one another directly in the sequence of services. Also, multiple pipe joins can be performed within a sequence of search services. As shown in Fig. 1, pipe joins are not represented by any dedicated symbol in query plans. Rather they are just a sequence of service invocations that are chained by passing the output of one invocation as input to the next.

4.2.2 Parallel Joins

Parallel joins enable parallelizing the invocation of Web services and are fundamental operations of query plans, where they are represented by means of dedicated nodes, shaped as a join symbol. Binary parallel joins have been studied in [3].

4.3 Invocation Strategy

Apart from the topology, a join method is characterized by the order and frequency in which the services involved in a join are invoked. We refer to this property as the *invocation strategy*. The choice of invocation strategy depends on the distribution of the ranking of the results and the cost of service invocation. We consider two cases named nested-loop and merge-scan, for their analogy to well-known join methods. In addition, other specific invocation strategies can be arbitrarily defined.

4.3.1 Nested-Loop

The nested loop strategy is suitable when the results of one search engine, conventionally the first service, exhibits a clear "step" (as defined in Section 4.1). In such case, we assume that the ranking of that service suddenly drops from a high value to a very low value. 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 h chunks corresponding to the high ranking values of the

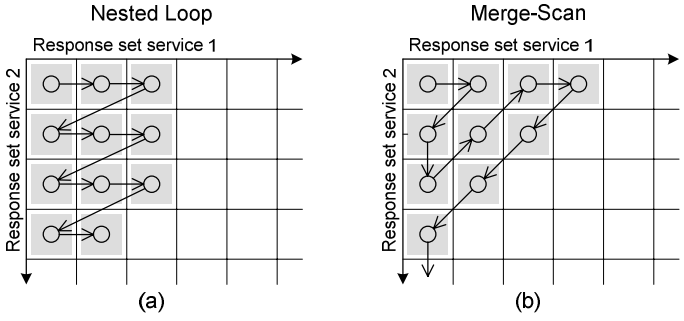

Fig. 5. Nested Loop (a) and Merge-Scan (b) strategies

"step" engine, and then extracting the chunks of the other service in ranking order, thereby producing join results. This strategy is represented in Fig. 5a.

4.3.2 Merge-Scan

The merge-scan strategy is indicated in the absence of information about a clear "step" in the ranking of results. Then, one should assume that rankings decrease progressively. 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 Fig. 5b, where the arrows indicate the order in which the tiles are chosen starting from the first tile. The method could evenly alternate service calls in the lack of better estimates of the score functions, or else it could use an **inter-service ratio** r between calls to services, and such ratio could be fixed (e.g. r=3/5) or variable. Chapter 11 presents top-k optimal join methods whose invocation strategy is merge-scan with variable inter-service ratios, based upon service costs. In Chapter 12 we show units for controlling the execution strategy, called clocks, whose function is to regulate service calls based upon the inter-service ratio.

4.4 Completion Strategy

Orthogonal to the invocation strategy that controls in which order and how often services are invoked, the *completion strategy* governs the order in which the tiles are considered by the join operation. Taken together, invocation and completion strategy thus control the exploration of the search space.

4.4.1 Rectangular

A rectangular strategy processes all the tiles as soon as the corresponding tuples are available. This completion strategy applies both to nested-loop and merge-scan. The rectangular strategy is locally extraction-optimal. With the nested loop method, if the step scoring function of the first service drops from 1 to 0 exactly in correspondence to the h-th chunk, then the method is globally extraction-optimal. A rectangular strategy matched to a particular sequence of requests is shown in Fig. 6.

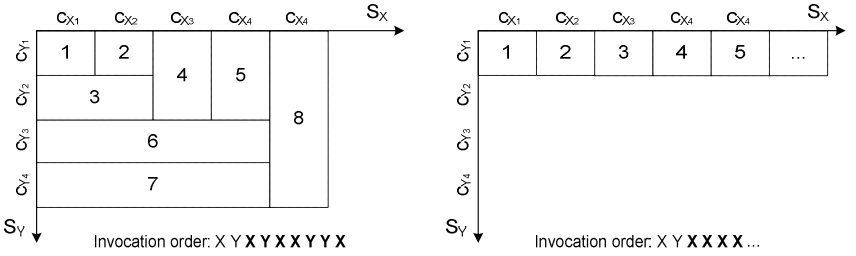

Fig. 6. Examples of rectangular completion strategies

It should be noted that a strong asymmetry in the ranking of the two services may lead to a "long and thin" rectangular completion strategy, 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 Fig. 6, has the disadvantage that each I/O only adds one tile.

4.4.2 Triangular

A triangular strategy processes all the tiles by moving "diagonally" in the Cartesian plan, as in the case of merge-scan, where a diagonal is expressed as ratio $r=r_1/r_2$ between numbers of blocks in the search space. Thus, the method processes tiles t_{xy} such that the sum of indexes of two consecutive tiles extracted by the strategy cannot increase by more than one and that $x\ r_2 + y\ r_1 < c$, where c are constant values which are progressively increased by the method, starting with $c= r_1\ r_2$. The triangular extraction strategy is locally extraction-optimal. When matched with the merge-join invocation strategy, it approximates an extraction-optimal strategy.

4.5 Join Methods

This classification—topology, invocation and completion strategy—gives rise to eight possible methods for the join of two services. Note that not all combinations that would be theoretically possible also make sense in practice.

As a very simple example of a method that makes sense, Fig. 7 shows a rectangular completion applied to a merge scan in which the inter-service ratio is fixed to 1, resulting in the exploration of squares of increasing size. This method typically makes sense for parallel joins in which chunks are fetched alternatively from the two joined services. Pipe joins are better performed via nested loops with rectangular completion, which corresponds to retrieving the same number of fetches from the second service for each invocation originating from each tuple in output from the first service.

An example of method that makes little sense in practice is a rectangular completion applied to nested loop.

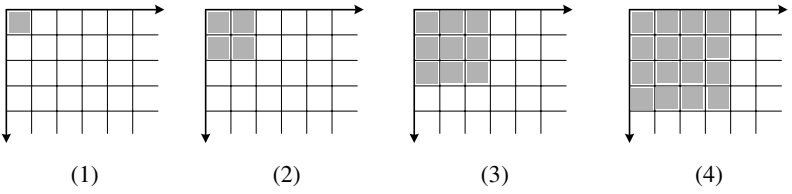

Fig. 7. Merge-scan, rectangular join strategy

5 Query Optimization for Search Computing

The optimization problem considered in this Section is: *given a query over a set of services, find the query plan that minimizes the expected execution cost according to a given cost metric in order to obtain the best k answers*. The process of generating an optimal plan starts from the conjunctive query over services (either service marts or service implementations) and ends with a fully instantiated invocation schedule.

5.1 Cost Metrics

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

Execution time metric, which measures the (expected) time elapsed from the query submission time to the production of the k-th answer. The time required for producing k tuples takes into account the number of invocations of each unit and the expected elapsed time for the execution of that unit in order to obtain a given number of results. The cost must account for the slowest path flowing tuples from the input to the output of the plan.

Sum cost metric, which computes the cost of a 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. A special case of the sum cost metric is the request-response cost metric, which consists of considering only the cost of service invocations required to execute the plan, omitting to consider operation execution costs. A further simplification is to assume that every service invocation has the same cost, and in such case the metric simply counts the number of calls. This metric is particularly relevant when the transfer of data over the network is the dominating cost factor.

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

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, extensively studied in [22], 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 the time required to present the user with the first output tuple. This metric is suitable for settings in which the user expects a prompt interaction.

5.2 Branch and Bound Approach to Query Optimization

After characterizing services, query plans, and cost metrics, we now introduce our optimization method, summarized in Fig. 8. The method explores the combinatorial solution space of all possible translations of the conjunctive query into fully instantiated invocation schedules. The exploration is organized by means of an incremental construction of the query plans, which takes place in three phases, imposing a discipline in the order in which alternative plans are generated and considered.

The first phase is the **selection of specific access patterns and service interfaces** for each service s_i occurring in the conjunctive query, so that the resulting query is proven as feasible[6]. This phase is required when the query is formulated at the highest level of abstraction, over service marts, and includes the selection of the "best" service interfaces. In the context of this chapter, service interfaces are already selected by the query, but feasibility has to be proven. If no feasible plan can be generated for a given query, the translation fails. Otherwise, the chosen service interfaces are passed to the second phase to set up the query topology.

The second phase is the **selection of a query topology** for the given choice of service interfaces. This phase fixes the order of invocation of the services, as well as the data flow and the details of join operations. Indeed, it is worth reminding that even when all access patterns have been determined, there may still be several alternative DAGs compatible with the precedence constraints they enforce.

The third phase is the **choice of the number of fetches** to be performed over the chunked services. This phase allows to fully determine the execution schedule and the join strategies, and therefore to compute its cost according to a given metric.

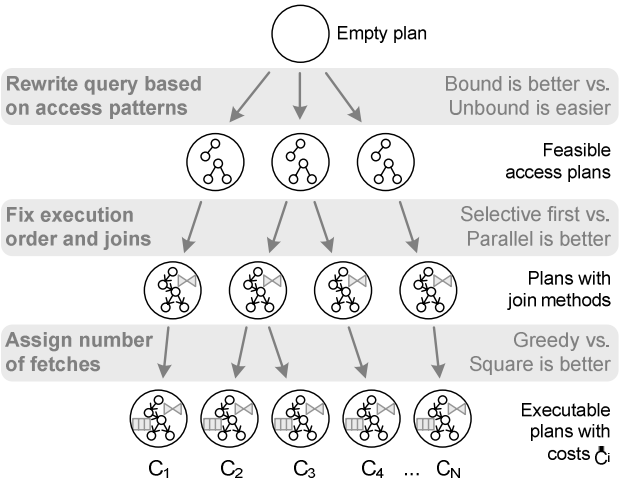

Fig. 8. Branch and bound

[6] Queries may also be formulated by means of syntax-aware user interfaces, such as the mashup environment described in [5]; in such cases, the query is guaranteed to be feasible by construction, as the tools do not allow users to compile unfeasible queries.

Each phase is combinatorial and the considered problem is hardly tractable by exact methods, even with queries involving few services. However, all considered cost metrics are monotonic, which allows for an exploration of the whole search space with a branch and bound strategy.

The incremental construction of query plans starts from an empty plan. Each choice in any of the three phases determines a subdivision of the search space into non-overlapping subsets, which is an ideal branching. Then, thanks to the mentioned monotonicity, each subset can be assigned a *lower bound* for the cost by calculating the cost on the partially constructed plan. To complete the *bounding* step, we can obtain an *upper bound* for a class of plans by fully constructing one plan 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 (and, implicitly, all solutions derivable from the elements of A) may be safely discarded from the search. In this way, the method converges to a local optimum, which under restrictive assumptions coincides with the global optimum.

This approach is traditionally 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. Our problem has a similar combinatorial explosion, and we have evidence (thorough prototype implementations, e.g., in [4]) that the optimization can find reasonably good solutions in acceptable execution time.

In the following, for each of the three phases we (i) define the branching and bounding steps to be used for examining the solution space, and (ii) describe heuristics for choosing the branches so as to build efficient plans quickly. The search for the optimal plan can be stopped at any time, and it will nevertheless return a valid solution. If let run up to exhaustion of the search space, the returned plan is the optimal one, otherwise the returned plan is the one representing the current upper bound, whose "distance" from the optimal one depends on the effectiveness of heuristic choices and the running time of the algorithm.

5.3 Phase 1: Access Pattern Selection

Query plans are to be constructed taking feasibility into account. Initially, all atoms in the query are considered. At least one atom must be reachable based on available access patterns for that atom, or else the query is not feasible. Selected atoms are associated to a service interface, which is either chosen by the system or by the user within all service interfaces with that access pattern. The association of the first atom to an access pattern corresponds to a family of access plans, obtained by setting all the first atom's attributes as bound, and therefore turning some other atoms as reachable. The process continues with branches driven by heuristics and bounds given by the cost of complete plans, unless some atom cannot be reached and the query is declared unfeasible. Lower bounds can be computed, e.g., by isolating the services that have fewer bound input attributes 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. The choice of the next atom can be done according to one of the following heuristics:

- **Bound is better:** a good heuristics for the choice of access patterns consists in preferring those with many input attributes. The intuition behind this heuristics is that the more attributes are bound to a given input, the smaller is the answer set, and therefore the service is faster in producing results, and less memory is required to cache the data.
- **Unbound is easier:** the shortcoming of the previous heuristics is that with many input attributes it is more difficult to find an assignment that makes the query feasible. Therefore, in contrast to the previous heuristics, an initialization with the minimum number of input attributes may make it easier to build a feasible solution.

5.4 Phase 2: Selection of a Query Topology

The construction of all possible DAGs for a query plan can be done incrementally. It starts by placing after the initial node some node corresponding to a reachable service, and then by progressively adding nodes corresponding to services that are reachable by virtue of the user input variables and the services already included in the query. Nodes can be added in series or in parallel with respect to already included nodes, compatibly with the constraints enforced by I/O dependencies. Clearly, the space of constructible DAGs may grow very quickly, due to the exponential number of choices, multiplied at each step of the construction. Yet, the number of choices also depends on the degrees of freedom on the partial order induced by the access patterns. indeed, if the access patterns determine a total order, then there is only one possible DAG. The choice of the next node in the query plan can be done according to one of the following heuristics.

- **Selective first:** having long linear paths in the DAG, ordered by decreasing selectivity, wherever possible (ideally, one chain from input to output).
- **Parallel is better:** 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 [22].

5.5 Phase 3: Choice of the Number of Fetches

Whenever a query includes chunked services, say $cs_1, \ldots, cs_M$, we need to provide an estimate of the number of chunks that will be fetched per input tuple at each cs_i. We call this numbers the *fetching factors* of the services, represented as a n-uple $\langle F_1,\ldots,F_M\rangle$. Initially, all fetching factors are set to 1, which is the lowest admissible value for such parameters, as all services must contribute to the result. Clearly, if the n-tuple $\langle 1,1,\ldots,1\rangle$ already determines $h \geq k$ results, then it is also the optimal solution. otherwise, the fetching factors have to be incremented, until $h \geq k$. This can be done incrementally, according to one of the following heuristics.

- **Greedy:** at each iteration, the F_i to be incremented is the one that corresponds to the node in the plan with the *highest sensitivity* with respect to the increase in the number of tuples in the query result *per cost unit*. Computing such

sensitivity parameter is rather complex as it takes into account the query topology, as the increase of tuples in output in one node causes more tuples to be processed (and hence costs) in all the successors of that node.

- **Square is better:** at each iteration, each F_i is incremented by a value that is proportional to its chunk size. This implies that, in average, after query execution, all chunked services will have explored about the same number of tuples. The name of the heuristics originates from the fact that the fetching factors are incremented in such a way that the explored parts of the search space of all binary joins are kept square and of the same size.

5.6 Optimization Applied to the Running Example

We now briefly show how optimization can be applied to the query of the running example. The query is already formulated over service interfaces, whose adornments are as follows:

Theatre_1 (Name^O, UAddress^I, UCity^I, UCountry^I, TAddress^O, TCity^O, TCountry^O, TPhone^O, Distance^R, Movie.Title^O, $\text{Movie.StartTimes}^O$, Movie.Duration^O)

Movie_1 (Title^O, Director^O, Score^R, Year^O, Genres.Genre^I, Language^O, $\text{Openings.Country}^I$, Openings.Date^I, Actor.Name^O)

Restaurant_1 (Name^O, UAddress^I, UCity^I, UCountry^I, RAddess^O, RCity^O, RCountry^O, Phone^O, Url^O, MapUrl^O, Distance^R, Rating^R, Category.Name^I)

With this choice of access patterns the query is feasible. referring back to the conjunctive formulation of Section 3.1, we note that all input attributes of Theatre and Movie are covered by INPUT variables, and the three input attributes of Restaurant are joined with the homonymous ones that are in output in Theatre. This test ends the first phase, identifying the I/O dependency that holds from Theatre to Restaurant. As for the second phase, four topologies are to be considered, shown in Fig. 9.

In all configurations Theatre precedes Restaurant, so as to implement with a pipe join the corresponding I/O dependency. Among the alternatives, we choose to continue our example with (d), which also contains a parallel join (as it would be chosen by the heuristics "parallel is better").

We then set K = 10 as the number of desired output combinations, and estimate the values of some basic parameters. Selectivity of the join predicates and chunk sizes are essential to calculate t^{in} and t^{out} for all the nodes in the plan. We estimate the selectivity of Shows() and DinnerPlace() as 2% and 40% respectively – namely, the probability that a given movie is being shown in a given theatre and the probability that a given theatre is placed close to a good restaurant. The value of K can be "back-propagated" through the nodes of the plan, as follows: K = 10 implies $t_{\text{Restaurant}}^{out} = 10$.

Fig. 9. Alternative topologies for the running example

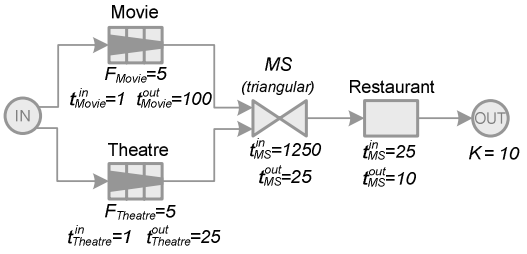

Fig. 10. Fully instantiated query plan for the running example

We choose to only keep and include in the result the first (and presumably best!) restaurant found for each location, therefore $t_{Restaurant}^{in} = 25$, by virtue of the selectivity of the pipe join. This in turn implies $t_{MS}^{out} = 25$, and therefore that the parallel join has to process 1250 candidate combinations overall. Here the space of possible solutions opens up quite widely, as different numbers of invocations and numbers of fetches per invocation can be assigned in order for 1250 combinations to be generated. Assuming that we restrict to the first 100 movies, corresponding to 5 fetches of chunks of 20 movies and to the first 25 theatres in order of distance from the user's address, corresponding to 5 chunks of size 5, we can consider the fully instantiated query plan obtained by annotating the plan of Fig. 10. Note that multiplying $t_{Movie}^{out} = 100$ by $t_{Theatre}^{out} = 25$ we obtain 2500, but choosing a triangular completion strategy assures that only the half of the "most promising" combinations (either close theatre or very good movies) are considered, thus obtaining $t_{MS}^{out} = 1250$.

We have only considered one possible instantiation of the query. It would be the branch and bound's responsibility to apply the cost metric to this "initial" plan, consider it as an upper bound, and explore the search space for less costly solutions.

6 Conclusions

This chapter has presented a formal model for the optimization and the execution of multi-domain queries over services. We have turned a high-level formulation of queries over services into executable query plans, describing the invocations of Web services and the composition of their inputs and outputs, and then we have presented a method for optimizing the selection of access plans according to cost metrics. Access plans include as its main ingredient the join between service results, and in this chapter we have reviewed efficient methods for join execution, which however do not guarantee that results are the top-k according to a global ranking. This chapter has therefore created the premises for the next two chapters, where we describe methods for rank aggregation which guarantee top-k results, and an engine for query plan execution.

References

1. Altinel, M., Brown, P., Cline, S., Kartha, R., Louie, E., Markl, V., Mau, L., Ng, Y.-H., Simmen, D., Singh, A.: Damia - A Data Mashup Fabric for Intranet Applications. In: VLDB 2007, pp. 1370–1373 (2007)

2. Bianchini, D., De Antonellis, V., Pernici, B., Plebani, P.: Ontology-based Methodology for e-Service Discovery. Inf. Syst. 31(4-5), 361–380 (2006)
3. Braga, D., Campi, A., Ceri, S., Raffio, A.: Joining the Results of Heterogeneous Search Engines. Inf. Syst. 33(7-8), 658–680 (2008)
4. Braga, D., Ceri, S., Daniel, F., Martinenghi, D.: Optimization of Multi-Domain Queries on the Web. PVLDB 1(1), 562–573 (2008)
5. Braga, D., Ceri, S., Daniel, F., Martinenghi, D.: Mashing Up Search Services. Internet Computing 12(5), 16–23 (2008)
6. Calì, A., Martinenghi, D.: Querying Data under Access Limitations. In: ICDE 2008, Cancún, Mexico, pp. 50–59 (2008)
7. Chamberlin, D.D., Astrahan, M.M., King, W.F., Lorie, R.A., Mehl, J.W., Price, T.G., Schkolnick, M., Selinger, P.G., Slutz, D.R., Wade, B.W., Yost, R.A.: Support for Repetitive Transactions and Ad Hoc Queries in System R. ACM-TODS 6(1), 70–94 (1981)
8. Confalonieri, R., Domingue, J., Motta, E.: Orchestration of Semantic Web Services in IRS-III. In: AKT-SWS 2004. The Open University, Milton Keynes (2004)
9. Daniel, F., Pernici, B.: Insights into Web Service Orchestration and Choreography. International Journal of E-Business Research 2(1), 58–77 (2006)
10. Deutsch, A., Ludäscher, B., Nash, A.: Rewriting Queries using Views with Access Patterns under Integrity Constraints. Theoretical Computer Science 371(3), 200–226 (2007)
11. DeWitt, D.J., Ghandeharizadeh, S., Schneider, D.A., Bricker, A., Hsiao, H.-I., Rasmussen, R.: The Gamma Database Machine Project. IEEE Trans. on Knowledge and Data Engineering 2(1), 44–62 (1990)
12. Florescu, D., Levy, A.Y., Manolescu, I., Suciu, D.: Query Optimization in the presence of Limited Access Patterns. In: SIGMOD 1999, Philadelphia, Pennsylvania, USA, pp. 311–322 (1999)
13. Ioannidis, Y.E., Kang, Y.: Randomized Algorithms for Optimizing Large Join Queries. SIGMOD Rec. 19(2), 312–321 (1990)
14. Ives, Z.G., Halevy, A.Y., Weld, D.S.: Adapting to Source Properties in Processing Data Integration Queries. In: SIGMOD 2004, Paris, France, pp. 395–406 (2004)
15. Kossmann, D., Stocker, K.: Iterative Dynamic Programming: a New Class of Query Optimization Algorithms. ACM-TODS 25(1), 43–82 (2000)
16. Li, C., Chang, E.: Answering Queries with Useful Bindings. ACM-TODS 26(3), 313–343 (2001)
17. Lohman, G.M.: Grammar-like Functional Rules for Representing Query Optimization Alternatives. In: SIGMOD 1988, Chicago, Illinois, USA, pp. 18–27 (1988)
18. Millstein, T.D., Levy, A.Y., Friedman, M.: Query Containment for Data Integration Systems. In: PODS 2000, Dallas, Texas, USA, pp. 67–75 (2000)
19. OASIS: Web Services Business Process Execution Language. Technical report (2007), `http://www.oasis-open.org/committees/wsbpel/`
20. Rajaraman, A., Sagiv, Y., Ullman, J.D.: Answering Queries using Templates with Binding Patterns. In: PODS 1995, San José, California, USA, pp. 105–112 (1995)
21. Seshadri, P., Hellerstein, J.M., Pirahesh, H., Cliff Leung, T.Y., Ramakrishnan, R., Srivastava, D., Stuckey, P.J., Sudarshan, S.: Cost-based Optimization for Magic: Algebra and Implementation. SIGMOD Rec. 25(2), 435–446 (1996)
22. Srivastava, U., Munagala, K., Widom, J., Motwani, R.: Query Optimization over Web Services. In: VLDB 2006, Seoul, Korea, pp. 355–366 (2006)

23. Özsu, M.T., Valduriez, P.: Principles of Distributed Database Systems. Prentice-Hall, Inc., Upper Saddle River (1991)
24. Tao, Y., Hristidis, V., Papadias, D., Papakonstantinou, Y.: Branch-and-Bound Processing of Ranked Queries. Inf. Syst. 32(3), 424–445 (2007)
25. Tatemura, J., Sawires, A., Po, O., Chen, S., Candan, K.S., Agrawal, D., Goveas, M.: Mashup Feeds: Continuous Queries over Web Services. In: SIGMOD 2007, New York, NY, USA, pp. 1128–1130 (2007)
26. W3C: Web Services Choreography Description Language, Version 1.0. W3C Candidate Recommendation (2005), http://www.w3.org/TR/ws-cdl-10/
27. W3C: Web Service Choreography Interface (WSCI), Version 1.0. W3C Note (2002), http://www.w3.org/TR/wsci/
28. WSMO: Web Service Modeling Ontology, http://www.wsmo.org
29. Yang, G., Kifer, M., Chaudhri, V.K.: Efficiently Ordering Subgoals with Access Constraints. In: PODS 2006, Chicago, Illinois, USA, pp. 183–192 (2006)
30. Yu, C.T., Meng, W.: Principles of Database Query Processing for Advanced Applications. Morgan Kaufmann, San Francisco (2005)