Chapter 6
Queries over Web Services

Efthymia Tsamoura, Anastasios Gounaris, and Yannis Manolopoulos

Aristotle University of Thessaloniki,
Thessaloniki, Greece
{etsamour,gounaria,manolopo}@csd.auth.gr

1 Introduction

Nowadays, technologies such as grid and cloud computing infrastructures and service-oriented architectures have become adequately mature and have been adopted by a large number of enterprizes and organizations [2,19,36]. A Web Service (WS) is a software system designed to support interoperable machine-to-machine interaction over a network and is implemented using open standards and protocols. WSs became popular data management entities; some of their benefits are interoperability and reuseability.

Seeking to benefit from the above opportunities, the web and grid data management infrastructures are moving towards a service oriented architecture by putting their databases behind WSs, thereby providing a well-documented, interoperable method of interacting with their data (e.g., [5,32]). Furthermore, data not stored in traditional databases can be made available via WSs. As a consequence, there is a growing interest in systems that are capable of processing complex queries (i.e., tasks) spanning services deployed on remote resources. The services can perform two operations; they either perform processing of data, or they play the role of a wrapper that retrieves data from a resource.

Currently, two classes of infrastructures that employ WSs to process data have been developed, namely the WS query infrastructures and the workflow management systems (WfMSs). The former process SQL-like queries or search queries over information sources (e.g. [5,4,42]). Like traditional database management systems, they perform the following tasks in order to answer a submitted query: query translation, service selection and query optimization. In the first two steps, the appropriate services that can correctly answer the submitted query are selected (either with or without user interaction), while the final step, which is the main topic of this chapter, aims to provide an efficient service execution plan. The other category comprises WfMSs, where the workflow components are services (e.g., [33,23]). In WfMSs the user has to select the services to process the data of interest, the location of the input data (which are either extracted by a service that polls a resource or they form a data stream) and the service invocation order, which is fixed. Languages such as BPEL4WS have emerged for specifying WS composition in workflow-oriented scenarios [1].

A. Vakali & L.C. Jain (Eds.): New Directions in Web Data Management 1, SCI 331, pp. 139–169.
springerlink.com

An example of a problem of optimization queries over WSs is given below. We assume that WSs provide an interface of the form $WS : X \rightarrow Y$, where X and Y are sets of attributes, i.e., given values for attributes in X, WS returns values for the attributes in Y, as shown in the following example adapted from [42]. In the generic case, the input tuples may have more attributes than X, while attributes in Y are appended to the existing ones.

Example 1. Suppose that a company wants to obtain a list of email addresses of potential customers selecting only those who have a good payment history for at least one card and a credit rating above some threshold. The company has the right to use the following WSs that may belong to third parties, the first of which contains a database of person ids.

$WS_1 : \emptyset \rightarrow SSN\ id\ \ (ssn)$
$WS_2 : SSN\ id\ \ (ssn, threshold) \rightarrow\ credit\ rating\ \ (cr)$
$WS_3 : SSN\ id\ \ (ssn) \rightarrow\ credit\ card\ numbers\ \ (ccn)$
$WS_4 : card\ number\ \ (ccn, good) \rightarrow good\ history\ \ (gph)$
$WS_5 : SSN\ id\ \ (ssn) \rightarrow\ email\ addresses\ \ (ea)$

There are multiple valid orderings to perform this task, although several precedence constraints exist: WS_1 must always be at the beginning and WS_3 must precede WS_4. The optimization process aims at deciding on the optimal (or near optimal) ordering under given optimization goals. When there are multiple logically equivalent services for the same task (e.g., there are two services containing email addresses at distinct places) or the physical placement of a service is flexible, then the problem becomes more complex. □

In this chapter we will discuss several different flavors of queries over WSs and the corresponding optimization algorithms. Note that some of these cases can be reduced to problems that have been examined in the context of traditional database queries in a straightforward manner. Traditional database solutions for such cases can be easily transferred to our setting by replacing database operators with WSs; for this reason, throughout the text, we will use the terms operators and services interchangeably. For example, the problem of optimal ordering of centralized WSs with a view to minimizing the response time may resemble the problem of ordering commutative filters in pipelined queries with conjunctive predicates [24,22], in the sense that the calls to WSs may be treated in the same way as expensive predicates. Note that ordering some types of relational joins can be reduced to the same problem, as well [7].

However, reducing the problem of optimizing queries over WSs to the problem of optimizing traditional queries is not always feasible because there are also many substantial differences, and, as such, several optimization problems encountered in queries over WSs have not been investigated in traditional query processing. These differences stem from the fact that, in queries over WSs, there may exist precedence constraints between the WSs, selectivities may be higher than 1 (e.g., WS_3 in the example can return more than one tuple) and, typically, the execution of queries over WSs typically takes place in a both distributed and parallel manner.

1.1 Optimization Problems of Queries over WSs

In this chapter, we examine several distinct query optimization problems that can be broadly classified into four main categories, namely operator ordering, operator scheduling, tuple routing and data transfer planning. For each problem, we present some of the known solutions. Note that these solutions are not directly comparable with each other since they deal with different problems.

Operator ordering where the goal is to build an operator (or WS) execution plan that minimizes a pre-defined criterion by defining an appropriate partial or total ordering of the operators. In other words, the optimization decisions relate to the ordering of operators in the execution plan exclusively and issues such as allocation of operators to resources do not apply. Note that the ordering need not be linear. Problems that fall into this category assume that necessary metadata (e.g., operator cost per input tuple, selectivity, etc.) are available and in addition, the operators are pre-allocated on host machines. A well-known problem is the min-cost operator ordering problem. Given a set of operators, the aim is to define an ordering of the operators so that all input queries are evaluated with the minimum total execution cost of operators. Optimization criteria will be discussed in more detail in Sec. 2.

Tuple routing which is a generalization of operator ordering in the sense that not only a single operator plan to be followed by all input tuples is created. The alternative approach, advocated by tuple routing techniques, may route input tuples through different plans, which are also termed as interleaving plans [13]. A set of interleaving plans consists of multiple simultaneously active operator plans, each of which processes different partitions of the original input tuple set. When a new tuple enters the system it is routed to one of these plans, according to a probability weight.

Operator scheduling where the goal is to decide the processor on which each service is evaluated. Problems of this category appear when the system is also responsible for resource allocation. It is assumed that the system is capable of performing dynamic service deployment before the execution of the query and there are multiple choices regarding the host nodes for each service. Operator scheduling can be examined either in conjunction with operator ordering or in isolation. In the latter case, operator ordering has been fixed in a previous step.

Data transfer planning where the focus is shifted to data transmission. The aforementioned query optimization problems are operation centric, i.e., they define the operator execution order and/or the operator location. In the data transfer problems, the primary concern is to optimize data transmissions. As such, these problems emphasize more on scheduling the data transmission operations, or on the specification of the amount of data exchanged between the hosts. Obviously, operator scheduling and data transfer planning problems are met only in parallel or distributed environments, whereas operator ordering and tuple routing problems are encountered in centralized settings, as well.

1.2 Chapter Contributions and Structure

The contribution of this chapter is twofold. First, it presents a detailed overview of the problems encountered in the optimization of queries over WSs. The problems do not differ only in their nature as detailed above, but also in regard to the type of queries, type of services or operators and the exact execution environment to which they are tailored. This discussion results in the development of a taxonomy-like classification of the problems in WS queries that appears in Sec. 2. Second, this chapter discusses state-of-the-art solutions to distinct flavors of the problem of optimizing queries over WSs in Sec. 3. Especially for the problem of minimizing the response time in decentralized pipelined queries, a novel algorithm is presented. A comparison of the key properties of the different solutions appears in Sec. 3.6, while Sec. 4 concludes the chapter.

2 Different Aspects of the Problem of Optimizing WS Queries

Before probing into advanced query optimization algorithms that are relevant to queries over WSs, we must first discuss the factors of the problem, which greatly affect its complexity. These factors refer to the execution environment, the type of input queries, the type of operators involved and the query optimization criteria and are common to all kinds of problems mentioned in Sec. 1. The taxonomy presented here aims at providing a complete view of these factors in a systematic way.

2.1 Execution Environment

We are mainly interested in queries in parallel and distributed query execution environments, like those in [35], since these environments are more common in WS queries. However, a great number of algorithms originally proposed for centralized environments are still relevant. In such single-processor systems, only a central node evaluates input queries, although the queries may process data from multiple distributed resources. A distributed environment, such as the Internet and the grid [19], consists of multiple, possible heterogeneous, independent and potentially autonomous sites that are loosely connected via a wide-area network. On the other hand, a parallel environment consists of multiple, homogeneous processors and data resources spread over a local network. As such, the communication cost may dominate the query execution process in a distributed environment, which, usually, is not the case in a parallel setting. Nevertheless, the similarities between parallel and distributed systems are more significant compared to their differences; so, we prefer to distinguish between centralized and non-centralized (i.e., either parallel or distributed) systems, only.

A parallel or distributed environment may be either static or dynamic. In the latter case, the environmental characteristics, such as the number of available processors, the processor workload, the network traffic, etc., may change over time rendering the problem of query optimization more challenging. Query optimization in dynamic environments, also called adaptive query processing [15], has been a topic of investigation since late 70s [18]; however, the problem has received renewed attention in the last decade. The vast majority of works on adaptive query processing, like those mentioned above, deal with changes in the operator characteristics and the input data rather than changes in the execution environment; only a few exceptions to this are known (e.g., [20]). Centralized environments are considered to be static; of course this is not always true, e.g., the amount of available memory may be subject to unpredictable changes, but the dynamicity of the environment can be safely overlooked when the resource characteristics we are mostly interested in, such as processing cost per tuple, usually play a minor role in optimization.

In addition, a distributed or parallel environment may utilize parallelism with a view to speeding up and scaling up query execution [17]. Three types of parallelism have been identified in parallel query processing, namely independent, partitioned and pipelined parallelism. In independent parallelism, query operators none of which use data produced by the others, may run simultaneously on distinct machines. In pipelined parallelism, data already processed by an operator may be processed by a subsequent operator in the pipeline, at the same time as the sender operator processes new data. Finally, partitioned parallelism refers to running several instances of the same operator on different machines concurrently, with each instance only processing a partition of the same original data set.

The three aforementioned forms of parallelism can co-exist within a single query execution plan. For instance, in the introductory example, WS_2 and WS_3 can process in parallel output data items of WS_1; this corresponds to independent parallelism. Also, WS_1 and WS_3 can be active simultaneously, i.e., WS_3 processes output tuples of WS_1, while the latter keeps generating new tuples; this corresponds to pipelined parallelism. Finally, consider a scenario where WS_3 is physically deployed

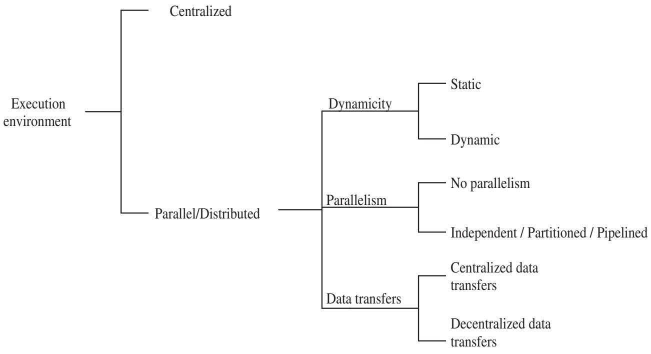

Fig. 1. Diagram of the different aspects regarding the execution environment

on two nodes, each processing half of the tuples of WS_1; in that case, partitioned parallelism is applied, as well. Obviously, parallelism can yield significant benefits only in multi-processor, parallel or distributed environments.

Distributed environments are also differentiated regarding the type of management of intermediate data transfers. In multi-processor systems, query operators can be placed and evaluated anywhere across the network. Regarding the intermediate data transfers, in a centralized data transfer approach, the intermediate data is transferred between resources via a central point. On the other hand, in a decentralized managed data transfer approach, processors exchange data directly. Since data is transferred from the source resource to the destination directly, the bottleneck problem caused in centralized approaches is ameliorated and these approaches are characterized by lower transmission times. Fig. 1 summarizes the different aspects regarding the execution environment.

2.2 Input Queries

The optimization algorithms that apply to queries over WSs either optimize each input query separately or optimize multiple queries simultaneously. In the latter case, they try to benefit from the overlap regarding the data resources they access, or even the constituent predicates. Multi-query optimization algorithms try to leverage this overlap in order to minimize the execution, communication and I/O cost. Additionally, input queries can be classified with respect to their time duration into continuous or non-continuous. Continuous queries are persistent queries that allow users to receive new results when they become available [43]. They are mainly met in streaming environments, where new data is continuously supplied and passed to WS sets for further processing. On the other hand, the non-continuous ad-hoc queries are executed on finite data. Optimization techniques that treat each tuple separately can be applied to both continuous and ad-hoc finite queries. An example of a continuous query over WSs is the following (adapted from [12]), where it is assumed that separate WSs are responsible for checking the price variations of Dell, Micron and Intel stocks:

"Notify me whenever the price of Dell or Micron stock drops by more than 5% *and the price of Intel stock remains unchanged over the next three months."*

Regarding their type, input queries can be expressed as traditional SQL-like database queries in the form of select-project-join (SPJ) and aggregates, or as search, information retrieval queries over information resources. Search queries are typically unstructured and often ambiguous; users submit one or more keywords to a search engine and the search engine returns approximate, i.e., incomplete answers with information that is related to the keywords provided in decreasing order of relevance. Fig. 2 provides a diagram of the different query aspects.

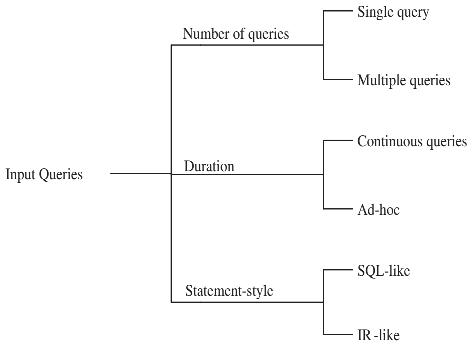

Fig. 2. Diagram of the different aspects regarding input queries

2.3 Input Operators

The type of input queries is also strongly correlated to the type of operators in the query execution plan. The operator attributes that are of interest include selectivity and precedence constraints (see Fig. 3). Selectivity is defined as the average ratio of output and input tuples. A WS that receives as input a country name and returns a list of major cities has average selectivity above one, and another service that, for the same input, returns just the capital has selectivity equal to one. Similarly, a service that may receive the name of any city in the world and returns airport codes only if the given city is nearby an airport has average selectivity below one, since, worldwide, there are fewer airports than cities. The operators in a query can be selective, i.e., their selectivity is between 0 and 1, or proliferative, i.e., their selectivity is greater than 1. IR-style services are typically characterized by high average selectivity values: given a single tuple containing a keyword, multiple data items are returned.

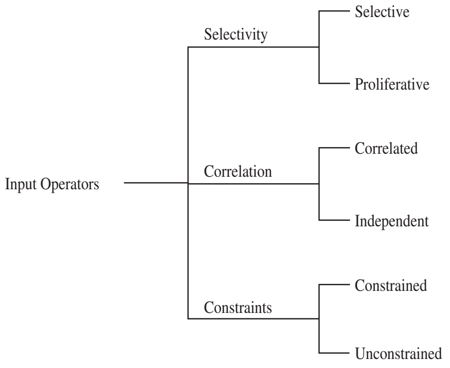

Fig. 3. Diagram of the different aspects regarding input operators

Moreover, the operators are considered to be correlated when their selectivity depends on the operators upstream in the query plan. If the selectivities are independent of the ordering, then the operators are called independent. Note that the selectivity of an independent operator may be correlated with the values of the input attributes, which is the case in [9]. A last parameter that categorizes the operators is the existence or not of prerequisite operators. More specifically, a constrained operator cannot be executed before the completion of its prerequisite operators, in contrast to an unconstrained one. The prerequisite operator O_j of an operator O_i is denoted by $O_j \prec O_i$.

2.4 Optimization Criteria

A critical factor in the optimization process is the exact optimization goal. Multiple criteria exist, such as maximizing query throughput, minimizing monetary cost, energy consumption, etc. In this chapter, we focus on two aspects, namely the minimization of the total query execution cost and the minimization of the query response time. Minimization of the total execution cost can be split in two parts. In centralized environments, execution cost encapsulates the cost for processing the operators and the disk I/Os. In distributed environments, the cost for transferring data among the operators is also considered. The minimization of execution cost aims at minimizing the sum of the processing and transmission cost for all operators in the query plan.

However, the opportunities imposed by parallelization have moved the interest to the optimization of other criteria, such as the response time, i.e., the time needed to produce the full result set. In a pipelined parallel environment, all operators process data simultaneously. As such, minimizing the query response time is equivalent to the minimization of the execution time of the longest running operator (often referred to as the bottleneck operator) instead of the sum of the execution times of all operators. When the query is evaluated with the help of interleaving plans (see Sec. 1.1), then the minimization of response time can be expressed as the maximization of the tuple flow [13]. These optimization criteria are depicted in Fig. 4.

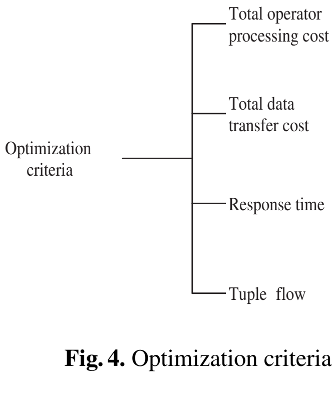

Fig. 4. Optimization criteria

3 Optimization Approaches

This section studies state-of-the-art algorithms for the problems presented in Sec. 1.1. The section starts by presenting some operator ordering problems in both static and adaptive execution environments and continues with tuple routing, scheduling and data transfer planning problems.

Independently of the execution environment, in the problems that are presented, except the data transfer planning ones, the data to be processed is either streamed by a single data resource or extracted from a database and then sent to subsequent services for processing. On the other hand, in the presented data transfer planning problems, we consider two different data resource models. In the first case, multiple data resources transfer data to a centralized processing component, while in the second case, data reside on traditional databases that are disparate across a network.

In order to answer a query involving calls to multiple services, the following actions must be performed by a query management component. First, the candidate services that may take place during the query execution phase must be selected. After that, statistics, regarding the per-tuple processing cost and the selectivity of the services, as well as the network status, must be gathered. This data is utilized by an optimization component that builds a feasible and efficient (in terms of a preselected optimization criterion) service ordering. It is assumed that the algorithms that deal with operator ordering and tuple routing problems exploit such query management components.

3.1 Operator Ordering Problems in a Static Environment

The operator ordering problems that are studied in the current subsection deal with static execution environments. In Sec. 3.1.1, we study problems, where the optimization objective is the query response time minimization, while in Sec. 3.1.2, we present problems, where the objective is the minimization of the per tuple total execution cost.

The services in Sec. 3.1 are considered to provide an interface of the form $WS : X \rightarrow Y$, where X and Y are sets of input and output attributes, respectively. Each WS typically performs operations such as filtering out data items that are not relevant to the query, transforming data items, or appending additional information to each input tuple.

3.1.1 Minimizing the Response Time

Srivastava *et al.* are among the pioneers that deal with query optimization when the data resources and the operators that process data are implemented as WSs. They consider a parallel and static execution environment, in which data is pipelined among services that are placed in arbitrary places. To this end, they propose a WSMS that, given an SQL-like input query, undertakes the task to produce an appropriate ordering of the services, in order to minimize the query response time. Query execution proceeds as follows. The output of one WS is returned to the WSMS and the latter redirects the received tuples to a subsequent WS, finally producing the query results. After giving a brief description of the execution environment, we present a formal problem definition utilizing the term operator instead of service.

More specifically, given an ad-hoc SPJ query Q that is defined over a set of N operators $O = \{O_1, O_2, \ldots, O_N\}$, the goal is to identify an operator ordering P for all input tuples that minimizes the response time of the query, in a parallel and static execution environment in which data is pipelined among the operators. Since the operators are executed in parallel, the maximum rate at which input tuples can be processed through the pipelined plan P is determined by the bottleneck operator (see Sec. 2.4). For every input tuple to P, the average number of tuples that an operator O_i needs to process is given by

$$R_i(P) = \prod_{k|O_k \in P_i(P)} \sigma_k \tag{1}$$

where $P_i(P)$ is the set of operators that are invoked before O_i in the plan P and σ_i is the service selectivity. The average processing time required by operator O_i per input tuple is $R_i(P)c_i$, where c_i is the per tuple processing cost of O_i. Since the cost of a plan is determined by the operator with the maximum processing time per input tuple, the bottleneck cost of a plan P is given by

$$cost(P) = \max_{1 \leq i \leq N} (R_i(P) \cdot c_i) \tag{2}$$

Srivastava *et al.* have proposed a greedy algorithm for the special case where the intermediate data transfers are centralized [42]. The operators are assumed to be independent, whereas arbitrary selectivity values and existence of precedence constraints are supported. In the produced plans, the output of an operator may be fed to multiple operators simultaneously. Starting from an empty operator plan, in every iteration of the algorithm, the next operator O_r to be appended to P is the one that incurs the minimum processing cost per tuple. In order to find the minimum cost of appending O_r to P, the best cut in P is found, such that on placing edges from the operators in the cut to O_r, the incurred cost is minimized. As such, the problem is reduced to a network flow problem [11]. The worst case complexity of the algorithm is $O(N^5)$ and the algorithm is provably optimal. For selective operators, the complexity is significantly lower since the optimal plan P is a linear ordering of the operators by increasing cost, ignoring their selectivity. For proliferative services, the produced plans may be parallel, i.e., a partial ordering is produced.

Example 2. Let $O = \{O_1, \ldots, O_{10}\}$ be a set of 10 operators with corresponding costs and selectivities shown in Table 1 and 2, respectively. Since all operators are selective, the proposed algorithm orders them by increasing processing cost. Thus, the optimal ordering that minimizes Eq.(2) according to [42] is $P = \{O_1 O_5 O_2 O_7 O_4 O_{10} O_8 O_9 O_3 O_6\}$. □

A drawback of this algorithm is that it does not take the potentially heterogeneous communication links between the operators into account. This is significant when the execution is decentralized, given also that the communication cost may be the

Table 1. Costs of operators in Example 2

O_i	1	2	3	4	5	6	7	8	9	10
c_i	2	7	12	8	4	16	7	10	10	9

Table 2. Selectivities of operators in Example 2

O_i	1	2	3	4	5	6	7	8	9	10
σ_i	0.8	0.7	0.9	0.3	0.5	0.6	0.4	0.1	0.6	0.7

dominant cost. In [42], it is assumed that the output of an operator is fed to the subsequent operators indirectly, through a central management component thus annihilating the need to consider the different communication costs explicitly. Tsamoura *et al.* address the afore-mentioned limitation by proposing a novel efficient algorithm for the optimal total ordering of operators, when the intermediate result transfers are decentralized and the communication costs between the operators may differ [46].

Let $t_{i,j}$ be the time needed to transfer a tuple from operator O_i to O_j. Similarly to [42], there is no limitation regarding the operator selectivities and the existence of precedence constraints; however, the selectivities are assumed to be independent, as well. The response time of a linear operator ordering S is given by the bottleneck cost metric in accordance to [42] with $t_{i,j}$ factored in:

$$cost(S) = \max_{1\leq i\leq N} R_i(S)(c_i + \sigma_i t_{i,i+1}), \tag{3}$$

where $t_{N,N+1} = 0$. $T_{i,j} = c_i + t_{i,j}\sigma_i$ is the aggregate cost of O_i with respect to O_j. The above formula implies that in general $T_{i,j} \neq T_{j,i}$, since c_i and σ_i values may differ from c_j and σ_j values. Note that if $t_{i,j}$ is equal for all service pairs, the problem can be solved in polynomial time, as shown in [42].

The proposed algorithm is based on the branch-and-bound optimization approach. It proceeds in two phases, namely the expansion and the pruning one. During expansion, new operators are appended to a partial operator ordering C, while during the latter phase, operators are pruned from C with a view to exploring additional orderings. The decision whether to append new operators or prune existing ones from a partial plan C is guided by two cost metrics, ε and $\overline{\varepsilon}$ respectively. The former corresponds to the bottleneck cost of C, and is given by Eq. (3), while the latter is the maximum possible cost that may be incurred by operators not currently included in C:

$$\overline{\varepsilon} = \max_{l,r} \left\{ \begin{array}{ll} \left(\prod_{j|O_j\in C}\sigma_j\right) T_{l,r}, & O_l \notin C,\ O_r \notin C \\ \left(\prod_{j=0}^{l-1}\sigma_j\right) T_{l,r}, & O_l :\ last\ operator\ in\ C,\ O_r \notin C \end{array} \right\} \tag{4}$$

The algorithm consists of the following simple steps. Starting with an empty plan C and an empty optimal linear plan S with infinity bottleneck cost, in every iteration of the algorithm, the parameters ε and $\overline{\varepsilon}$ are computed. If the bottleneck cost ε of

C is lower than $\overline{\varepsilon}$, then a new operator is appended to C; this operator is the one having the minimum aggregate cost with respect to the last operator in C. If the bottleneck cost ε of the current plan C is higher than or equal to the bottleneck cost ρ of the best plan found so far S, then the operators in C after the bottleneck service, including the latter, are pruned. Finally, whenever the condition $\overline{\varepsilon} \leq \varepsilon < \rho$ is met, a new solution is found. That condition implies that the ordering of the services that are not yet included in C does not affect its bottleneck cost. As a consequence, all plans with prefix the partial plan C have the same bottleneck cost. So, a candidate optimal solution S is found that consists of the current plan C followed by the rest of the services in any order. The bottleneck cost ρ of the best plan found so far S is set to $\rho = \varepsilon$. The last solution is the optimal one. The detailed description of the algorithm, along with the proofs of correctness and optimality can be found in [46]. Furthermore, detailed real-world ([45]) and simulation ([46]) evaluation has shown that the proposed algorithm can yield significant performance improvements (of an order of magnitude in many cases).

The following example demonstrates the steps of the algorithm proposed in [46] for minimizing the response time in a distributed and static environment, which employs pipelining during query execution.

Example 3. Let us assume that the operators in Example 2 are allowed to communicate directly with each other and the network connections are heterogeneous. The corresponding aggregate costs of the operators are shown in Table 3. For example, the cell $T_{1,2}$ of Table 3 is evaluated as $T_{1,2} = c_1 + t_{1,2}\sigma_1 = 2 + 65 * 0.8$, where 0.8 is the per cost to transfer an tuple from O_1 directly to O_2.

Fig. 5 shows the partial plans at the end of each iteration. Initially, the plans C and S are empty and the bottleneck cost of S is set to ∞. The algorithm starts by identifying the operator pair, which incurs the minimum bottleneck cost. The corresponding operators are O_1 and O_7. After that, $C = O_1 O_7$. In the second iteration, since $\varepsilon = 8 < \overline{\varepsilon} = \sigma_1 \times T_{7,3} = 36$ and $\varepsilon < \rho = \infty$, a new operator is appended to C, the one having the minimum aggregate cost with respect to O_7; that operator is O_9.

Table 3. Aggregate cost matrix **T**

i \ j	1	2	3	4	5	6	7	8	9	10
1	-	54	35	42	14	50	8	33	17	10
2	52	-	18	33	47	40	69	37	42	43
3	49	26	-	60	68	74	98	40	66	57
4	23	19	24	-	17	46	21	9	42	27
5	11	33	35	19	-	10	40	52	14	32
6	52	44	57	91	23	-	44	22	72	46
7	10	43	45	24	36	26	-	35	17	19
8	14	15	14	11	20	11	17	-	17	16
9	21	40	46	78	22	66	25	48	-	79
10	16	45	44	53	48	44	29	47	90	-

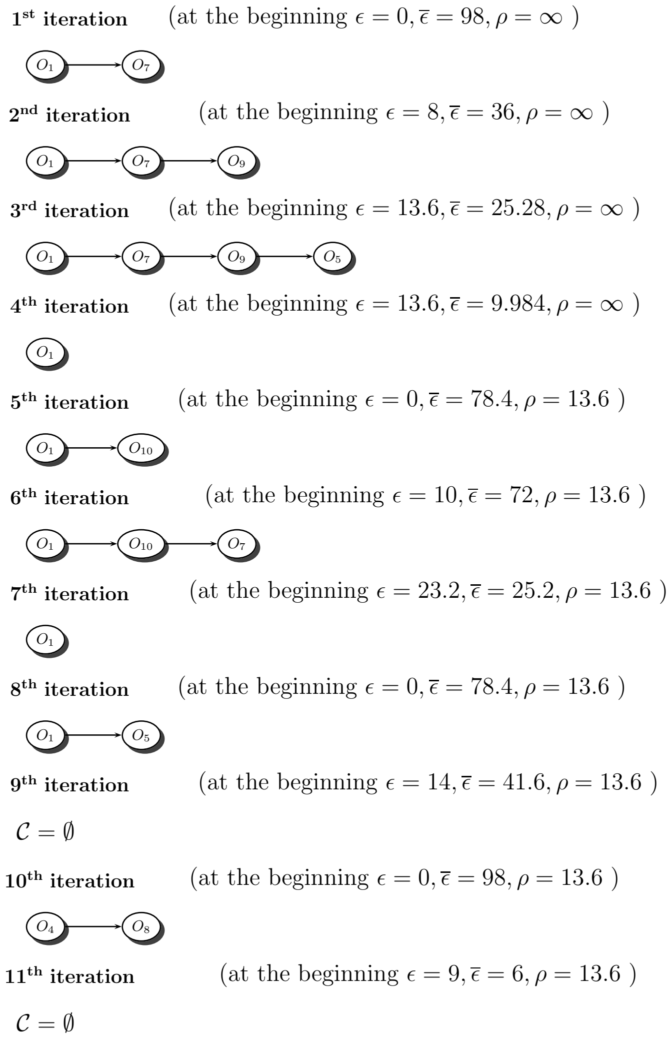
1st iteration (at the beginning $\epsilon = 0, \overline{\epsilon} = 98, \rho = \infty$)
O_1
O_7
2nd iteration (at the beginning $\epsilon = 8, \overline{\epsilon} = 36, \rho = \infty$)
O_1
O_7
O_9
3rd iteration (at the beginning $\epsilon = 13.6, \overline{\epsilon} = 25.28, \rho = \infty$)
O_1
O_7
O_9
O_5
4th iteration (at the beginning $\epsilon = 13.6, \overline{\epsilon} = 9.984, \rho = \infty$)
O_1
5th iteration (at the beginning $\epsilon = 0, \overline{\epsilon} = 78.4, \rho = 13.6$)
O_1
O_{10}
6th iteration (at the beginning $\epsilon = 10, \overline{\epsilon} = 72, \rho = 13.6$)
O_1
O_{10}
O_7
7th iteration (at the beginning $\epsilon = 23.2, \overline{\epsilon} = 25.2, \rho = 13.6$)
O_1
8th iteration (at the beginning $\epsilon = 0, \overline{\epsilon} = 78.4, \rho = 13.6$)
O_1
O_5
9th iteration (at the beginning $\epsilon = 14, \overline{\epsilon} = 41.6, \rho = 13.6$)
$\mathcal{C} = \emptyset$
10th iteration (at the beginning $\epsilon = 0, \overline{\epsilon} = 98, \rho = 13.6$)
O_4
O_8
11th iteration (at the beginning $\epsilon = 9, \overline{\epsilon} = 6, \rho = 13.6$)
$\mathcal{C} = \emptyset$

Fig. 5. The steps in Example 3

In the third iteration, since $\varepsilon = 13.6 < \overline{\varepsilon} = \sigma_1 \times \sigma_7 \times T_{9,10} = 25.28$ and $\varepsilon < \rho = \infty$ the operator O_5 is appended to C forming the partial plan $C = O_1 O_7 O_9 O_5$. Now, since $\varepsilon = 13.6 > \overline{\varepsilon} = \sigma_1 \times \sigma_7 \times \sigma_9 \times T_{5,8} = 9.984$, and $\varepsilon < \rho = \infty$, a solution is found. Thus, S is set to C, $\rho = 13.6$ and C is pruned. After the pruning, $C = O_1$ (the bottleneck operator is O_7). The termination condition, see [46], is not triggered given that there exists a two operator prefix that has not been investigated and its cost is less than ρ: $T_{4,8} = 9$.

In the fifth iteration, since $\varepsilon = 0 < \overline{\varepsilon} = 78.4$ and $\varepsilon = 0 < \rho = 13.6$, a new operator is appended to $C = O_1$; that is O_{10}. A new operator is also appended in the sixth iteration forming the partial plan $C = O_1O_{10}O_7$. In the seventh iteration, the partial plan is set to $C = O_1$, as $\varepsilon = 23.2 > \rho = 13.6$ and the bottleneck operator is the second one, i.e., O_{10}. In the eight iteration, O_5 is appended to $C = O_1$, while in the ninth iteration, the partial plan is set to $C = \emptyset$, as $\varepsilon = 14 > \rho = 13.6$ and the bottleneck operator is the first one, i.e., O_1. As a result, any other plan starting with O_1 can be safely ignored. Since the plan C is empty, the algorithm searches for the pair of operators with the minimum aggregate cost. In our example, this pair consists of O_4 and O_8. In the eleventh iteration, a new solution is found, since $\varepsilon = 9 > \overline{\varepsilon} = 6$ and $\varepsilon < \rho = 13.6$. Thus, $S = O_4O_8$, the bottleneck operator is O_4 and ρ is set to 9. After the pruning $C = \emptyset$, and the algorithm safely ignore plans starting with O_4O_8. This causes the algorithm to terminate, since the cost of the less expensive operator pair except those beginning with O_1, which is O_5O_6, is higher than ρ: $T_{5,6} = 10 > \rho = 9$. So the algorithm terminates, after having essentially explored all the 10! orderings in just 11 iterations. □

The characteristics of these two state-of -the-art algorithms for the optimization of queries over WSs in a pipelined parallel environment are summarized in Table 4.

Table 4. Operator ordering algorithms for minimizing the response time

Work	Execution environment	Input queries	Input operators
[42]	Parallel/distributed, static, centralized data transfers, pipelined parallelism	Single, ad-hoc, SQL-like	Independent, both selective and proliferative, both constrained and unconstrained
[46]	Distributed, static, decentralized data transfers, pipelined parallelism	Single, ad-hoc, SQL-like	Independent, both selective and proliferative, both constrained and unconstrained

3.1.2 Minimizing the Total Processing Time

In previous sections we saw that the query response time equals the maximum execution cost spent by an operator in order to process an input tuple and/or to send them to a subsequent operator. On the other hand, in a min-cost operator ordering problem, the goal is to minimize the total operator execution cost (processing and or transferring) that is incurred per input tuple. From now on, the term execution cost, unless clarified otherwise, encapsulates both the processing and transferring cost spent by an operator.

Ordering operators with a view to minimizing the per tuple total execution cost, a problem also commonly referred to as the min-cost operator ordering problem, is essential for achieving good system throughput. In general, solutions to the min-cost problem initially proposed for single-node settings may be applied to parallel settings characterized by resource homogeneity in a straightforward manner.

The min-cost ordering problem comes in several flavors. One of the most interesting ones refers to a parallel and static execution environment, where data is directly exchanged between the operators through pipelining; data communication can occur via a coordinator as well, without essentially modifying the problem, as long as homogeneous network links are assumed. If the operators are independent, then well-established fast solutions apply (e.g., [24,22]). However, correlated operators pose a more challenging problem. More specifically, given an ad-hoc select query Q that is defined over a set of unconstrained, correlated and selective operators $O = \{O_1, O_2, \ldots, O_N\}$ with fixed processing cost c_i and selectivity σ_i, the goal is to find an operator linear ordering S that minimizes the total execution cost of operators per input tuple. This cost encapsulates only the processing cost of tuples and is formally given by the following equation:

$$cost(S) = c_1 + \sum_{i=2}^{N} c_i D_i,\; D_i = \prod_{j=1}^{i-1}(1 - d(j|j-1)) \tag{5}$$

$d(i|j)$ is the conditional probability that the operator O_i will drop a tuple that has not been dropped by any of the operators that precede O_i in $\mathscr{S}$, and $d(i|0) = 1 - \sigma_i$ is the unconditional probability that operator O_i will drop a tuple. Any drop probability is linearly related to selectivity, given that $d(i|j) = 1 - \sigma(i|j)$. Babu *et al.* have proved that this problem is equivalent to the pipelined set cover problem [7]. The pipelined set cover problem is MAX SNP-Hard [30], which implies that any polynomial operator ordering algorithm can at best provide a constant-factor approximation guarantee for this problem. In [30], a 4-times approximation algorithm is introduced to solve this problem. According to that algorithm, the operators must be ordered in a way that satisfies the following condition (termed greedy invariant):

$$\frac{d(i|i-1)}{c_i} \geq \frac{d(j|i-1)}{c_j}, 1 \leq i \leq j \leq N \tag{6}$$

Example 4. We continue Example 2, aiming now at minimizing the total execution time (only the processing time is considered). In this example, the selectivities of the operators are independent, so the algorithm in [7] is reduced to those in [24,22] and the operators are ordered in decreasing order of $(1 - \sigma_i)/c_i$. As such, first O_5 is selected, followed by O_1, O_8 and so on. If the operators were correlated, then, after selecting O_5, the conditional selectivities $\sigma(i|5)$ of all other operators would have to be estimated, in order to detect the second operator. □

Next, the min-cost operator ordering problem is studied in a multi-query setting. This problem is also known as the shared min-cost operator ordering problem. More formally, let $Q = \{Q_1, Q_2, \ldots, Q_M\}$ be a set of M, potentially continuous select queries that are evaluated over a set of N selective, unconstrained and correlated operators O. Each query is a conjunction of the operators in O. For each input tuple, operator $O_i \in O$ either returns a tuple or rejects it. The proportion of rejected tuples is defined by the operator selectivity. The goal is, given an input tuple t, to find the ordering that identifies the queries satisfied by t with the minimum cost. Note that an input tuple satisfies a query if it is not rejected by none of its constituent operators.

Obviously, if a query is satisfied, then all of its constituent operators must be evaluated. On the other hand, if an operator of a query rejects a tuple, then we do not have to evaluate the rest operators belonging to the same query (and are not evaluated so far). Thus, in a produced ordering, only a subset of operators $O' \subseteq O$ have to be evaluated, in order to determine the queries that are satisfied, and thus, the per tuple total processing cost is given by:

$$cost(S) = \sum_{i|O_i \in O'} c_i \tag{7}$$

Munagala *et al.* have proved the equivalence of this problem to the minimum set cover problem, and proposed an approximate greedy algorithm as a solution [31]. More formally, at any stage of the algorithm, the next operator to be evaluated is expected to resolve the maximum number of unresolved queries per unit cost. We say that for a given tuple t, a query is resolved if all its constituent operators return t or the currently evaluated operator rejects t. Let p_i be the number of unresolved queries the operator O_i is part of and $1 - \sigma_i$ the probability that the operator O_i rejects an input tuple. Then, the expected number of queries resolved by O_i is $p_i(1 - \sigma_i)$. The next operator to be evaluated is the one that minimizes the ratio $rank_i = c_i/p_i(1-\sigma_i)$. O_i is then removed independently from filtering out or not an input tuple. In addition, the queries that have been resolved due to the operator evaluation, and any other operator, which is not part of at least one not yet resolved query, is also removed. The algorithm terminates when all submitted queries are resolved.

Example 5. This example presents the steps of the algorithm proposed by Munagala *et al.* for the shared min-cost operator ordering problem. Suppose that the following queries are available $Q_1 = \{O_1, O_3, O_7, O_{10}\}$, $Q_2 = \{O_4, O_5, O_8\}$, $Q_3 = \{O_2, O_4, O_{10}\}$, $Q_4 = \{O_1, O_7, O_9, O_{10}\}$, $Q_5 = \{O_3, O_6, O_7, O_9\}$. Let t be an input tuple, which is not rejected by any operator (except O_1, O_5 and O_9). As a consequence, only the query Q_3 is satisfied for t, since none of its constituent operators rejects this tuple. Figure 6 shows the results after every iteration of the algorithm. The algorithm starts by identifying the operator which minimizes the ratio $rank_i = c_i/p_i(1-\sigma_i)$, which is O_7 with $rank_7 = 3.88$ (see also Tables 1 and 2). Since O_7 does not reject the input tuple, no query is resolved. After that, operator O_1 is selected with $rank_1 = 5$. Operator O_1 rejects t, thus resolving queries Q_1 and Q_4. None of the not yet evaluated operators is removed, since they are included in at least one of the not yet resolved queries, i.e., Q_2, Q_3 and Q_5. The next two operators are O_4 and O_5 with $rank_4 = 5.71$ and $rank_5 = 8$, respectively. Since O_5 rejects t, query Q_2 is also resolved. The next operator is O_2 with $rank_2 = 23.33$. After evaluating operator O_2, operator O_9 is evaluated with $rank_9 = 25$. Since O_9 rejects input tuple t, query Q_5 is also resolved. Apart from that, operators O_3 and O_6 do not have to be evaluated, since they are not part of any unresolved query. Finally, the remaining operator, i.e., O_{10} is evaluated, and thus query Q_3 is satisfied. □

Liu *et al.* have proposed an edge-coverage-based approximate greedy algorithm for the same problem that achieves a better approximation ratio [28]. In [31], the shared min-cost operator ordering problem is viewed as the problem of covering the input

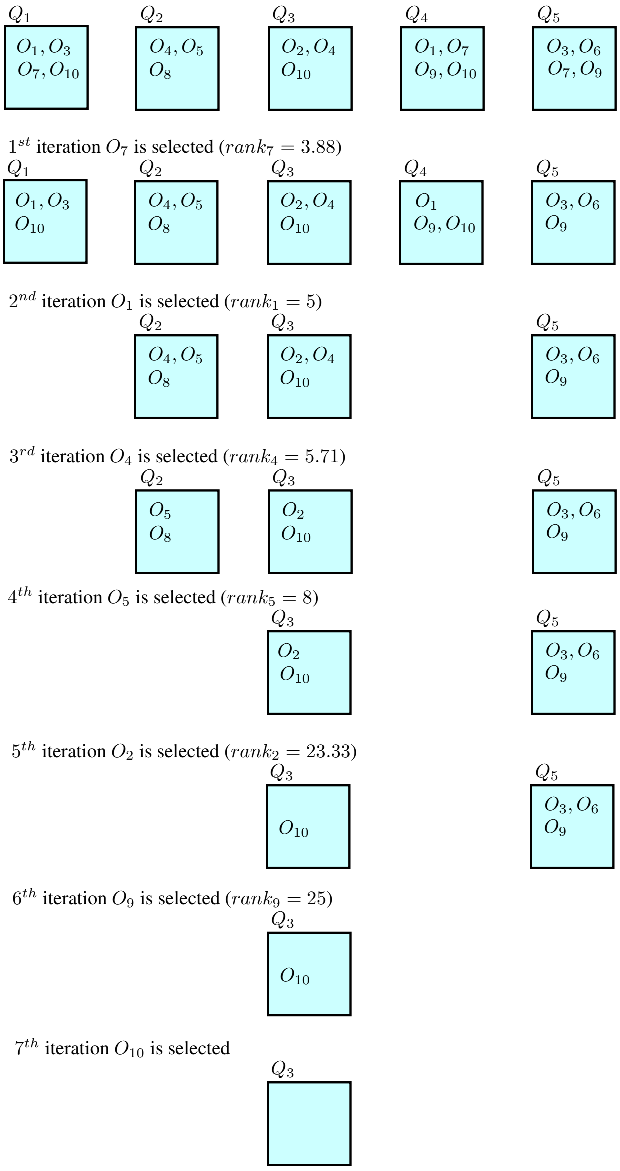
Q_1 O_1, O_3 O_7, O_{10}
Q_2 O_4, O_5 O_8
Q_3 O_2, O_4 O_{10}
Q_4 O_1, O_7 O_9, O_{10}
Q_5 O_3, O_6 O_7, O_9
1^{st} iteration O_7 is selected ($rank_7 = 3.88$)
Q_1 O_1, O_3 O_{10}
Q_2 O_4, O_5 O_8
Q_3 O_2, O_4 O_{10}
Q_4 O_1 O_9, O_{10}
Q_5 O_3, O_6 O_9
2^{nd} iteration O_1 is selected ($rank_1 = 5$)
Q_2 O_4, O_5 O_8
Q_3 O_2, O_4 O_{10}
Q_5 O_3, O_6 O_9
3^{rd} iteration O_4 is selected ($rank_4 = 5.71$)
Q_2 O_5 O_8
Q_3 O_2 O_{10}
Q_5 O_3, O_6 O_9
4^{th} iteration O_5 is selected ($rank_5 = 8$)
Q_3 O_2 O_{10}
Q_5 O_3, O_6 O_9
5^{th} iteration O_2 is selected ($rank_2 = 23.33$)
Q_3 O_{10}
Q_5 O_3, O_6 O_9
6^{th} iteration O_9 is selected ($rank_9 = 25$)
Q_3 O_{10}
7^{th} iteration O_{10} is selected
Q_3

Fig. 6. The steps in Example 5, where each box corresponds to an unresolved query containing its remaining operators

queries through a suitable choice of operators. However, in [28], the same problem is viewed as the problem covering the connections between queries and operators through a suitable choice of operators, rather than covering the queries themselves.

The algorithm makes use of a bipartite graph. A bipartite graph $G = (O, Q, E)$ consists of two partitions, the set of the not yet evaluated operators (initially O) and the set of the not yet resolved queries (initially Q); an edge $e = (O_i, Q_j) \in E$ between an operator and a query indicates the fact that the operator is present in the corresponding query. Given an input tuple t, the next operator to be evaluated in each step is the one that covers the maximum number of edges in the bipartite graph with the minimum processing cost. For an operator O_i, the expected number of edges covered is the sum of the expected number of the queries that O_i evaluates to true[1] plus the expected number of the operators that do not have to be evaluated if operator O_i evaluates to false (these are the not yet evaluated operators that belong to the not yet evaluated queries, where O_i is part of). More formally, the next operator to be evaluated, given a tuple t, is the one that minimizes the ratio

$$unit - price_i = \frac{c_i}{\sigma_i \delta(O_i) + (1 - \sigma_i) \sum_{\forall Q_k | (O_i, Q_k) \in E_i} \delta(Q_k)} \tag{8}$$

where E_i is the remaining set of edges in the current iteration, δ is the degree of an operator or a query respectively in the bipartite graph and Q_k is any query involving O_i. After each operator evaluation, the bipartite graph is updated with the performed actions being identical to those in [31].

Example 6. We reconsider the problem in Example 5 employing the edge-coverage based algorithm proposed in [28]. Figure 7 shows the operator-query bipartite graph after every iteration of the algorithm. Let t be the current input tuple. In the first iteration, operator O_1 is selected for evaluation with unit-price = $2/(0.8*2+0.2*(4+4)) = 0.625$. After that, queries Q_1 and Q_4 are removed from the graph along with operator O_1, since O_1 rejects t. In the second iteration, the operator O_4 is selected with unit-price = $8/(0.3*2+0.7*(3+3)) = 1.66$. In the third iteration, O_7 is selected, while in the fourth iteration we evaluate O_5 with unit-price = $4/(0.5+0.5*2) = 2.66$. Since operator O_5 rejects the input tuple, query Q_2 is removed from the graph along with operator O_8. The latter is removed as it is not part of any unresolved query. In the fifth iteration, O_2 is selected for evaluation, while in the sixth iteration, O_9 is selected with unit-price=$10/(0.6+0.4*3) = 5.55$. After evaluating operator O_9, query Q_5 along with operators O_3 and O_6 are removed from the graph. Finally, operator O_{10} is evaluated, since it is the only remaining operator. □

A common problem with the performance of WSs is that they may be too slow or prohibitively expensive in some cases. In that case, if there exist some additional highly selective operators that are inexpensive and correlated to the expensive ones, it is beneficial to incorporate them early in the plan. This is the main rationale in

[1] We say that, given a tuple t, an operator O_i evaluates to true a query Q_i if $O_i \in Q_i$ and O_i returns t. Otherwise, we say that O_i evaluates Q_i to false.

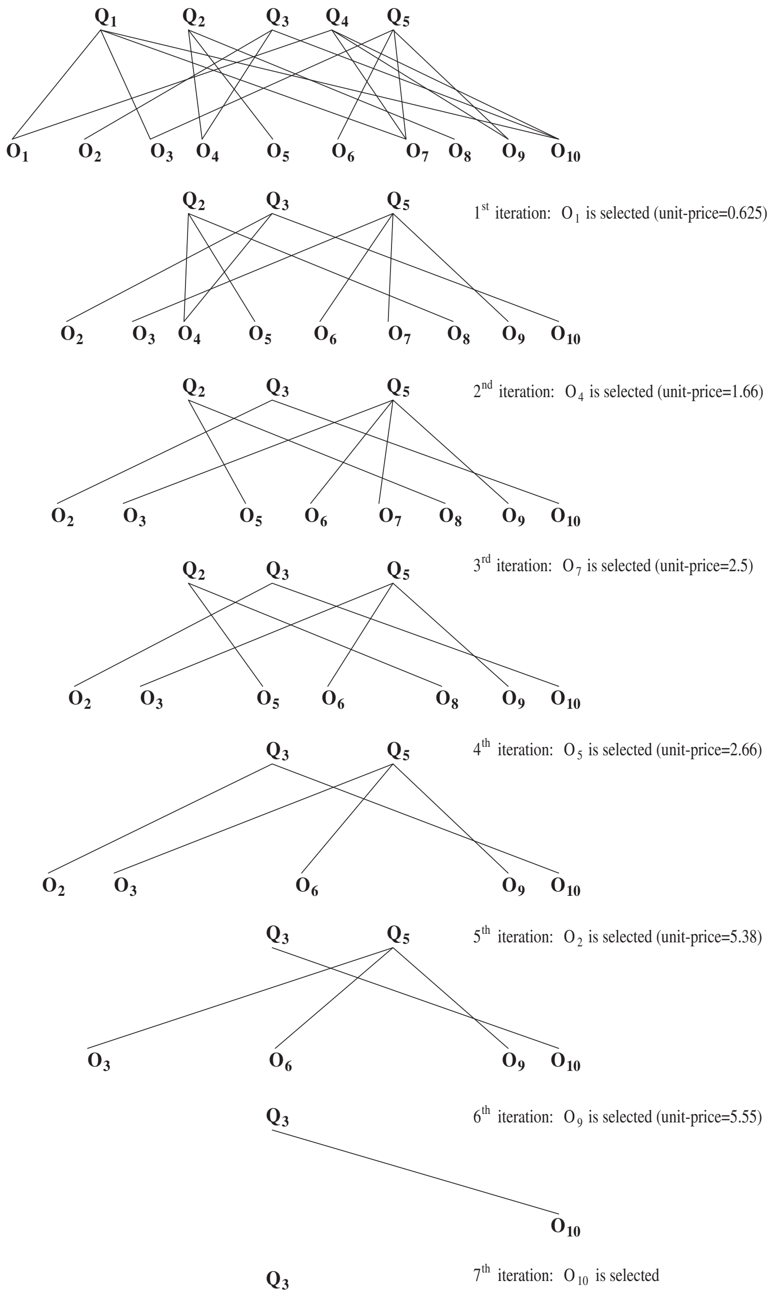
Q_1 Q_2 Q_3 Q_4 Q_5
O_1 O_2 O_3 O_4 O_5 O_6 O_7 O_8 O_9 O_{10}
1st iteration: O_1 is selected (unit-price=0.625)
2nd iteration: O_4 is selected (unit-price=1.66)
3rd iteration: O_7 is selected (unit-price=2.5)
4th iteration: O_5 is selected (unit-price=2.66)
5th iteration: O_2 is selected (unit-price=5.38)
6th iteration: O_9 is selected (unit-price=5.55)
7th iteration: O_{10} is selected

Fig. 7. The steps in Example 6

[14]. More specifically, the work in [14] exploits the fact that some additional low-cost operators $O_i \notin O$ can be evaluated so as to reject tuples at lower cost avoiding the cost of evaluating expensive operators. In this work, the proposed solution is a conditional plan. A conditional plan is a decision tree where each interior node corresponds to an operator that splits the plan into several conditional plans, in turn. During evaluation, the tree is traversed. At every node, the query processor evaluates the corresponding operator and follows one of the sub-plans depending on its output.

Lazaridis and Mehrota in [25] deal with a problem similar to [14]. Let Q be an ad-hoc, select query evaluated over a set of N independent, unconstrained and selective operators O. The goal is to find an operator ordering that minimizes the total operator processing cost per tuple, in order to decide whether an input tuple is rejected by any of the operators or not. In [25], as in [14], this is done by inserting additional lower cost operators that are not part of the original operators mentioned in the query explicitly.

A motivation example is as follows. Suppose a query that uses an expensive face identifier method, which compares input images with stored images containing faces of criminals:

*SELECT * FROM Camera, Criminals*
WHERE FaceIdentifier(Camera.image, Criminals.image)

Suppose also that the user has access to two additional, less expensive methods, ObjectDetector and FaceDetector which detect foreign objects and human faces, respectively. By using these methods we can reject some input images at lower cost: if an image does not contain an object or a face, then there is no reason to test whether it contains a particular face of a criminal. We infer a negative result for FaceIdentifier from a negative result by either of the two methods ObjectDetector and FaceDetector.

Table 5 summarizes the main characteristics of the proposed solutions to different flavors of the min-cost operator ordering problem. As already explained, although some of these solutions were originally proposed for centralized settings,

Table 5. Algorithms for flavors of the min-cost operator ordering problem

Work	Execution environment	Input queries	Input operators
[7]	Parallel, decentralized data transfers, pipelined parallelism	Single, continuous, SQL-like	Correlated, selective and unconstrained
[31]	Centralized	Multiple, continuous, SQL-like	Correlated, selective and unconstrained
[28]	Centralized	Multiple, continuous, SQL-like	Independent, selective and unconstrained
[14]	Centralized	Single, ad-hoc, SQL-like	Correlated, selective and unconstrained
[25]	Centralized	Single, ad-hoc, SQL-like	Independent, selective and unconstrained

their results can be easily transferred to parallel settings or distributed settings with centralized data transmission, and, as such, they can be employed to optimize queries over potentially remote WSs. Also, techniques proposed for continuous queries may be applicable to ad-hoc queries on finite data, too.

3.2 Operator Ordering Problems in Dynamic Environments

Wide area settings hosting WSs are typically subject to changes, which may have significant impact on queries. Babu *et al.* have extended their work in [7] to address the more general problem where the execution environment is dynamic. This is achieved by utilizing two components, a so-called "profiler" and a "re-optimizer". The profiler maintains a time-based sliding window of tuples dropped in the recent past. A profile tuple is created for every tuple in the sliding window and shows which operators have unconditionally rejected it. The re-optimizer can then compute any selectivity estimates that it requires from the profile tuples. The re-optimizer's job is to ensure that the current operator ordering satisfies Eq. (6). Similarly, the operator costs can be monitored, as well. The above render the algorithm proposed in [7] robust to environmental changes.

In the context of adaptive query processing [15], Avnur and Hellerstein have proposed the eddies execution model for minimizing the response time of ad-hoc SPJ queries at runtime [6]. The operators can be of arbitrary type, i.e., both selective and proliferative, both constrained and unconstrained, and both correlated and independent. In the eddies model, every tuple may follow a different plan. The original eddy implementation employed two main approaches to routing. The first one, called back-pressure, causes more tuples to be routed to fast operators early in query execution. The second approach augments back-pressure with a ticket scheme, whereby the eddy gives a ticket to an operator whenever it consumes a tuple and takes a ticket away whenever it sends a tuple back to the eddy. In this way, higher selectivity operators accumulate more tickets. When choosing an operator to which a new tuple should be routed, the ticket-routing policy conducts a lottery between the operators, with the chances of a particular operator winning being proportional to the number of tickets it owns. In this way, higher selectivity operators tend to receive more tuples early in their path through the eddy. The algorithm in [7] can also be incorporated into eddies routing policies. Several extensions to eddies have been proposed, including the works in [9,38,29,44]. The work of Tian and DeWitt [44] explicitly considers distributed execution environments supporting decentralized data transferring. In a distributed eddy, each operator, instead of returning processed tuples back to a central eddy, it redirects them to a subsequent operator. The operators learn query execution statistics and exchange them with other operators periodically. Based on these statistics, each operator makes its own routing decisions without consulting the central eddy or any other operator. By employing such eddies in distributed queries over WSs, the need of an optimizer that constructs an execution plan becomes obsolete.

3.3 Tuple Routing and Scheduling Problems

WSs can usually process many requests concurrently due to multi-threading. Each server hosting a WS has limited capacity though, so an optimizer has to build plans that respect the capacity constraints. Allowing multiple concurrent calls to operators is considered in tuple routing problems. These problems deal with the selection of one or more operator orderings, in order to maximize the number of tuples processed per unit time, this is why they are also called flow maximization problems. Their main rationale is to benefit from as much capacity of the processors hosting the operators as possible.

Condon *et al.* have proposed a solution for the special case where the orderings are linear, the execution environment is parallel and static, the data transfers are decentralized and pipelined parallelism is employed [13]. Let Q be an ad-hoc select query consisting of calls to N independent, unconstrained and selective operators O. r_i is the rate limit of each operator and is measured in tuples per time unit. Each tuple can be routed individually, so that different tuples can follow distinct routes. The problem is to find one or more operator orderings in order to maximize the tuple flow per unit time. More formally, suppose that a set of M different linear operator orderings are available, $\{\pi_1, \pi_2, \ldots, \pi_M\}$ that process different subsets of input tuples in parallel. Let f_i be the number of tuples sent through linear plan π_i per unit time. Then, the total number of tuples processed per unit time by the different linear orderings is given by

$$F = \sum_{\pi_i} f_i, \quad f_i > 0 \,\forall\, \pi_i \tag{9}$$

The goal is to find the set of f_i and π_i values that maximize Eq. (9) without violating the rate limits r_i of the operators.

Condon *et al.* proposed a recursive algorithm for this problem, which is detailed in [13]. Operators are initially ordered (from N to 1) in a way that satisfies the following condition:

$$r_i \sigma_i \leq r_{i+1} \forall i, 1 \leq i \leq N-1, \tag{10}$$

After that, the flow of tuples along each ordering is increased until either (i) an operator is saturated, i.e., it processes the maximum possible number of input tuples according to its rate limits, or (ii) the residual capacity of O_i, $1 \leq i \leq N$ times its selectivity σ_i becomes equal to the residual capacity of O_{i+1}. In [13], it is shown that if stopping condition (i) is satisfied, the constructed flow is optimal. On the other hand, if stopping condition (ii) is satisfied, the operator O_{i+1} is immediately placed after O_i and the operators O_{i+1} O_i are replaced by a single operator $O_{i,i+1}$ with rate limit equal to the residual capacity of O_i and selectivity equal to the product $\sigma_i \sigma_{i+1}$. The resulting smaller problem is then solved recursively.

Example 7. Let $O_2 = \{O_1, O_2, O_3, O_4, O_5\}$ be a set of five operators with rate limit and selectivity values $\{12, 8, 7, 4, 2\}$ and $\{0.9, 0.3, 0.7, 0.5, 0.8\}$, respectively. Initially, the operators are sorted in descending rate limit order, i.e., O_3 O_4 O_2 O_5 O_1; this ordering satisfies Eq. (10). The minimum flow of tuples that triggers either of the two conditions (see [13]) is $f_{3,4,2,5,1} = 4.36$. If $f_{3,4,2,5,1} = 4.36$ tuples per time unit are sent through the ordering O_3 O_4 O_2 O_5 O_1, then the residual capacity of O_2 times its selectivity is equal to the residual capacity of O_4. After that, the ordering O_3 O_4 O_2 O_5 O_1 is kept and a new operator ordering is created. To this end, operators O_2 and O_4 are merged into a single operator with residual capacity $r_{2,4} = 5.82$ (the residual capacity of O_2) and selectivity $\sigma_{2,4} = \sigma_2 \times \sigma_4$. The new smaller sub-problem is solved recursively. In the second iteration, $f_{3,2,4,5,1} = 5.25$ is the minimum flow of tuples that triggers stoping condition (2), while none of the operators becomes saturated with less flow. Thus, the ordering O_3 $O_{2,4}$ O_5 O_1 is kept and the operators $O_{2,4}$ and O_5 are merged into an operator $O_{5,2,4}$ with residual capacity equal to the residual capacity of O_5 and selectivity $\sigma_{5,2,4} = 0.105$, forming a new ordering O_3 $O_{5,2,4}$ O_1. The problem is again solved recursively. The algorithm terminates in the fifth iteration, where a single operator $O_{1,5,2,4,3}$ has been left with residual capacity $r_{1,5,2,4,3} = 0.2316$, i.e., $f_{1,5,2,4,3} = 0.2316$ tuples per time unit must be sent along this ordering, in order to saturate the single operator. Together, the flows constructed in the aforesaid five stages yield the following optimal solution to the max-throughput tuple routing problem for the given input instance: $f_{3,4,2,5,1} = 4.36$, $f_{3,2,4,5,1} = 5.25$, $f_{3,5,2,4,1} = 1.58$, $f_{3,5,2,4,1} = 1.58$, $f_{3,1,5,2,4} = 0.79$, $f_{1,5,2,4,3} = 0.2316$ and $f_\pi = 0$ for all other π orderings. The steps of the algorithm are shown in Figure 8. □

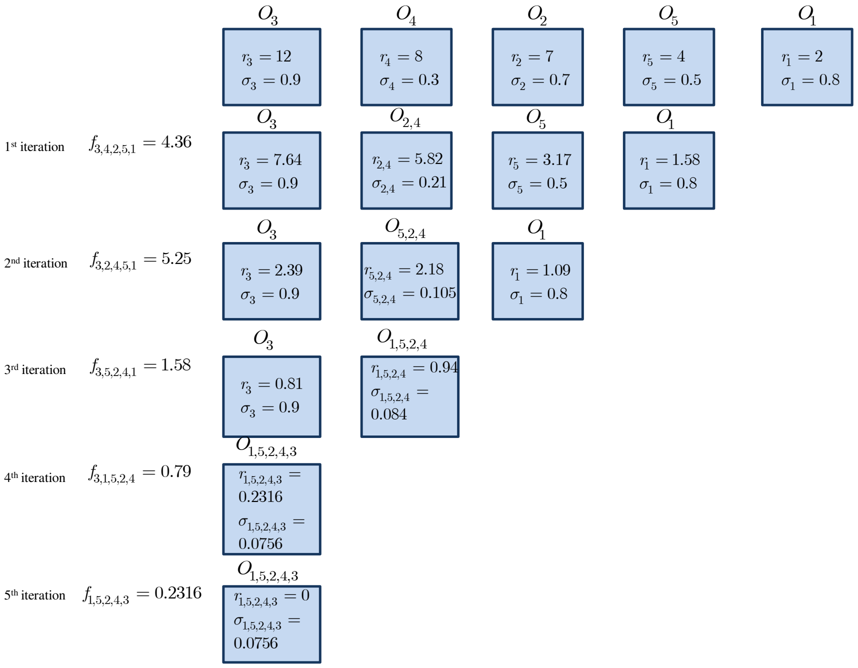
O_3
O_4
O_2
O_5
O_1
$r_3 = 12$ $\sigma_3 = 0.9$
$r_4 = 8$ $\sigma_4 = 0.3$
$r_2 = 7$ $\sigma_2 = 0.7$
$r_5 = 4$ $\sigma_5 = 0.5$
$r_1 = 2$ $\sigma_1 = 0.8$
1st iteration
$f_{3,4,2,5,1} = 4.36$
O_3
$O_{2,4}$
O_5
O_1
$r_3 = 7.64$ $\sigma_3 = 0.9$
$r_{2,4} = 5.82$ $\sigma_{2,4} = 0.21$
$r_5 = 3.17$ $\sigma_5 = 0.5$
$r_1 = 1.58$ $\sigma_1 = 0.8$
2nd iteration
$f_{3,2,4,5,1} = 5.25$
O_3
$O_{5,2,4}$
O_1
$r_3 = 2.39$ $\sigma_3 = 0.9$
$r_{5,2,4} = 2.18$ $\sigma_{5,2,4} = 0.105$
$r_1 = 1.09$ $\sigma_1 = 0.8$
3rd iteration
$f_{3,5,2,4,1} = 1.58$
O_3
$O_{1,5,2,4}$
$r_3 = 0.81$ $\sigma_3 = 0.9$
$r_{1,5,2,4} = 0.94$ $\sigma_{1,5,2,4} = 0.084$
4th iteration
$f_{3,1,5,2,4} = 0.79$
$O_{1,5,2,4,3}$
$r_{1,5,2,4,3} = 0.2316$ $\sigma_{1,5,2,4,3} = 0.0756$
5th iteration
$f_{1,5,2,4,3} = 0.2316$
$O_{1,5,2,4,3}$
$r_{1,5,2,4,3} = 0$ $\sigma_{1,5,2,4,3} = 0.0756$

Fig. 8. The steps in Example 7

[16] extends the algorithm in [13] so that precedence constraints among operators and arbitrary selectivities are supported.

Liu *et al.* deal with the joint problem of flow maximization and scheduling in a heterogeneous, multi-query, parallel-processor environment [27]. More formally, let $Q = \{Q_1, Q_2, \ldots, Q_K\}$ be a set of K ad-hoc select queries and $O = \{O_1, O_2, \ldots, O_N\}$ be a set of N, potentially correlated, unconstrained and selective operators. Each query is a conjunction of the operators in O. For every input tuple t, operator $O_i \in O$ either returns a tuple or rejects it. Furthermore, let M be a set of heterogeneous processors and $y_i(M_k)$ be the cost of evaluating operator O_i on processor M_k. The goal is to find, for each input tuple t, one or more operator orderings and an associated allocation scheme so that the flow of tuples processed per unit time is maximized. In the proposed algorithm, each operator O_i is always evaluated on the processor M_k for which the incurred load $y_i(M_k)$ is minimized (for simplicity we denote $y_i(M_k)$ by y_i). For the operator ordering problem, Liu *et al.* leverage the fact that the flow maximization problem in heterogeneous processing environments is equivalent to the problem of min-cost operator ordering in centralized environments, where the cost of each operator O_i is given by y_i. The problem addressed is a generalized case of the shared min-cost operator problem introduced in [31] and [28]. Other operator scheduling problems are studied in [47,8,41,3,37].

The main characteristics of the algorithms are summarized in Table 6.

Table 6. Solutions to tuple routing problems (first row) and scheduling problems (second row)

Work	Execution environment	Input queries	Input operators
[13,16]	Parallel, static, decentralized data transfers, pipelined parallelism	Single, ad-hoc, SQL-like	Independent, both selective and proliferative, both constrained and unconstrained
[27]	Parallel, static, heterogeneous processors, pipelined parallelism	Multiple, ad-hoc, SQL-like	Both independent and correlated, both selective and unconstrained

3.4 Data Transfer Planning Problems

Consider a dynamic environment, where multiple data sources stream data to a central processing node through heterogeneous communication links, in order to evaluate aggregate queries. These queries combine data from multiple data sources and their answers must be re-computed as data updates arrive to the sources. It is assumed that each data source stores the values of a single data attribute. The goal is to minimize the total communication cost, in order to evaluate multiple (possibly overlapping) aggregate continuous queries. No parallelism is employed during queries execution.

Olston *et al.* have provided a solution for this problem exploiting the fact that the precise answer of a continuous query may not always be necessary [34]. In such cases, approximate answers of sufficient precision may be computed from a small fraction of the input stream items. Users need to submit quantitative precision constraints along with continuous queries, which the processing node uses to filter stream items at the remote data sources. Each query is associated with a pair of real values, L and H that define an interval $[L\,H]$ in which the precise answer is guaranteed to lie. The reasoning behind the algorithm is quite simple: a data source does not need to stream the data that does not affect the answer, according to the previously defined precision requirements. For example, if the current exact answer is 10 and the precision interval is [7 13], then data sources holding updated data with values from 7 to 13 do not have to proceed to data transmission. The heuristic algorithm for filtering the stream items on the remote resources, called filter tuning, consists of an iterative two-step procedure. In the first step, each data resource shrinks the bound width periodically at a predefined rate. Each time the bound width of a data source shrinks, the so-called "leftover" width is reallocated to other data sources, ensuring all precision constraints are still satisfied. In the second step, the data sources that increase their bound widths are heuristically selected; the algorithm selects the ones that stream data at high rates and are connected with expensive communication links.

Li *et al.* in [26] deal with another data transfer planning problem. In this work, it is assumed that the data resources are spread across a wide-area distributed environment, and there is a single data resource per host. The links between hosts are heterogeneous, while the data resources can directly transfer data to other resources. Any SQL-like query can be submitted, and there is no limitation regarding the type of operators. For every submitted query, a query plan is provided in the form of a rooted tree. The plan specifies the operators to be evaluated on each data resource and the evaluation order. The aim is to schedule the data transfers across the queried data resources, in order to minimize the total data transferring cost when evaluating the query plans. It is proven that the problem is NP-hard for arbitrary communication networks by a reduction from the Steiner tree problem in graphs [39]. Li *et al.* proposed a polynomial time algorithm for this problem, which relies on the weighted hyper-graph minimum cut algorithm [26]. The produced data movement plan is optimal for tree-shaped communication networks, while it is an approximation to the optimal one for more general communication networks.

3.5 Other Problems Related to Queries over WSs

Thus far we have dealt with problems in which the services provide exact answers. Search queries belong to a different paradigm. In [10], Braga *et al.* deal with the joint problem of finding a WSs plan and an access pattern for each service for search query optimization. The execution environment is distributed and static, while the services can exchange data directly through pipelining. The proposed algorithm explores the space of plans using a heuristic, branch and bound strategy and it is applicable, under some modifications, for the optimization of both the total service processing cost and the response time criteria. Another common problem with WSs is that they are slow and the communication cost may well dominate the processing cost. Block-based data transmission may alleviate this problem, as proposed in [42,21]. Finally, [40] explore adaptive approaches to parallelizing calls to WSs.

3.6 Discussion and Open Issues

The purpose of the current section is to summarize the problems that have been studied for query optimization over WSs and the state-of-the-art algorithms that have been proposed.

Concerning the operator ordering problem in static execution environments, we have presented several algorithms that aim to minimize either the response time of the submitted query, or the per tuple total processing cost. Srivastava *et al.* introduced a general purpose WSMS for query optimization in a parallel environment that utilizes pipelined parallelism during query execution [42]. The major assumptions that are made are the following. The services do not exchange data directly, but a central component undertakes the intermediate data transfers. Also, the selectivity and the processing cost of the operators are constant and independent of the input attribute values. Under these assumptions, a provably optimal algorithm has been proposed that schedules parallel invocations of the operators, in order to minimize the response time of the submitted queries. The work of Tsamoura *et al.* comprises an extension of the work in [42] for distributed execution environments, where the operators exchange data directly over non-negligible and heterogeneous communication links [46]. The pipelined execution model is also applied. However, the proposed algorithm builds only linear operator orderings. A problem of significant importance that has not been addressed is the generalization of the latter algorithm for building parallel operator invocations. Furthermore, it would be very interesting to study the query response time minimization problem in a dynamic environment, where the per tuple processing and transferring costs, change significantly over time.

After that, we have presented several flavors of the min-cost ordering problem both in centralized [31,28,14,25] and parallel execution environments [7]. The above min-cost operator ordering problems deal with select queries, while the cost needed to transfer tuples from one service to another is negligible. In [7], given an input query that is evaluated through a set of correlated, unconstrained and selective operators, the goal is to build a linear operator ordering that minimizes the total cost of processing operators per input tuple.The proposed algorithm provides a 4-times approximation solution, while an heuristic technique has been introduced for the generalization of the above algorithm when the wide-area settings are subject to changes. Furthermore, Munagala *et al.* [31] and Liu *et al.* [28] deal with a multi-query flavor of the min-cost ordering problem in a centralized execution environment. In particular, given a set of one or more queries that consist of a set of independent, selective and unconstrained operators, the goal is to find an optimal operator ordering, in order to answer all input queries with the minimum total processing cost. The last two works that are studied try to minimize the per tuple total processing cost for an input query by utilizing additional, lower cost and highly selective operators [14,25]. Those operators need not be part of the initial operator set. To summarize, for operator ordering in a static execution environment, the following problems have been addressed:

- Response time minimization of single SPJ queries employing pipelined parallelism and independent operators both in a parallel and distributed execution environment. Decentralized data transfers have been considered, as well.
- Total operator execution cost minimization. Three different problem flavors that consider unconstrained operators are discussed; namely, (i) optimization of a single-query that employs parallelism and assumes correlated operators in a parallel environment, (ii) optimization of a single query with correlated operators in a centralized environment both for correlated and independent operators and (iii) optimization of multiple queries with both independent and correlated operators in a centralized environment.

Regarding the response time minimization, no work has been done for multi-query optimization or correlated operators. Furthermore, other types of parallelism (such as partitioned) have not been considered. The above works deal with SQL-like queries. The only work that deals with IR-like queries is presented in [10]. [10] deals with the joint problem of selecting the more appropriate services to invoke, when multiple services have the same functionality but different binding patterns, and of ordering the selected services in a distributed and static execution environment that employees pipelined parallelism. However, no performance guarantees have been provided.

In the context of adaptive operator ordering for minimizing the response time of a query, eddies [6] and distributed eddies [44] try to overcome the “hassle” of varying processing and communication costs in a dynamic execution environment by routing each tuple independently.

In Sec. 3.3, we have presented two throughput maximization problems [13,27]. Both of them employ inter-operator parallelism in order to maximize the tuple throughput, i.e., the number of tuples processed by the operators per unit time. The work in [16] deals with single query optimization, imposing only the independent assumption on input operators. On the other hand, the work in [27] deals with multiple select queries and unconstrained, selective operators. It also performs operator scheduling. For both proposals, the underlying execution environment is static and parallel. In general, adaptive query processing is in its infancy.

We have dedicated the last part of Sec. 3 to the description of two data transfer planning problems in a static and dynamic execution environment. Both of them deal with multiple input queries, while the communication links are heterogeneous. The work of Olston *et al.* deals with a centralized execution environment, where multiple continuous aggregate queries are evaluated in a central processing component [34]. There the data resources are disparate in a wide-area network and periodically stream data to the central component. Considering that the precise answer is not always necessary for some or all input queries, the goal is to appropriately tune the amount of data sent by each remote data resource, in order to minimize the total communication cost for answering input queries. Li *et al.* deal with another data transfer planning problem. As in [26], the data resources are disparate in a wide-area heterogeneous environment, while the latter can directly exchange data. Given a plan that specifies the operations to be performed on input data the goal is to appropriate schedule the data exchanges among the resources, in order to minimize the total data transferring cost when evaluating the input queries. Both works do not encapsulate the processing cost in order to answer the submitted queries. A limitation of the problem considered in [34] is that it handles only aggregate queries, while a limitation of [26] is that it requires a plan that specifies the operator invocation order. As both works deal with the min-cost metric, an interesting aspect would be the exploration of problems having other optimization criteria, such as the response time of the submitted queries. For example, regarding the problem in [26], the objective might be to minimize the maximum response time of the submitted queries. The characteristics of the proposed algorithms are summarized in Tables 4, 5 and 6.

The following remarks arise from the above discussion. The data transfer cost and the network heterogeneity issue are largely overlooked. In the majority of the works, the state-of-the-art operator ordering and tuple routing algorithms consider parallel and/or centralized execution environments, where the processing cost dominates. Another important issue that needs more attention is dynamicity. The presented problems mainly deal with static execution environments which is not the case in wide-area infrastructures such as the grid. The operator independence assumption must be reconsidered, since, in a real setting, the processing cost and the selectivity of utilized operators may be tightly related with the input attributes values. Another problem that should be investigated in the future is the combination of different forms of parallelism. Finally, the majority of presented problems deal with SQL-like queries. Extending current approaches or investigating new ones for IR queries optimization is crucial, since querying information sources is an important part of information management in Web and other distributed wide-area organizations.

4 Conclusion

This chapter discussed queries over WSs focusing on their optimization. Queries over WSs are becoming increasingly common due to the proliferation of publicly available WSs and remote and decentralized computing infrastructures such as grid and cloud computing. We presented a taxonomy of the problems encountered in the optimization of such queries taking into account the type of the optimization problems, the type of queries, the type of services or operators and the exact execution environment to which the queries are tailored. Some of the problems can be efficiently solved by utilizing known algorithms from the database community, whereas, for some others, novel algorithms have been proposed. This chapter discussed the state-of-the-art solutions that apply to the problem of optimizing queries over WSs, explaining their main characteristics. Especially for the problem of minimizing the response time in decentralized pipelined queries, a novel algorithm was presented.

References

1. Business process execution language for web services,
   `http://bpel.xml.org/tags/bpel4ws`
2. Abadi, D.J.: Data management in the cloud: Limitations and opportunities. IEEE Data Eng. Bull. 32(1), 3–12 (2009)
3. Agrawal, K., Benoit, A., Dufossé, F., Robert, Y.: Mapping filtering streaming applications with communication costs. In: Proc. of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 19–28 (2009)
4. Aloisio, G., Cafaro, M., Fiore, S., Mirto, M., Vadacca, S.: Grelc data gather service: a step towards P2P production grids, pp. 561–565 (2007)
5. Alpdemir, M.N., Mukherjee, A., Gounaris, A., Paton, N.W., Watson, P., Fernandes, A.A.A., Fitzgerald, D.J.: Ogsa-dqp: A service for distributed querying on the grid. In: Proc. of the International Conference on Extended Database Technologies (EDBT) (2004)
6. Avnur, R., Hellerstein, J.M.: Eddies: continuously adaptive query processing. In: Proc. of the International Conference on Management of Data (SIGMOD), pp. 261–272 (2000)
7. Babu, S., Matwani, R., Munagala, K.: Adaptive ordering of pipelined stream filters. In: Proc. of the International Conference on Management of Data (SIGMOD), pp. 407–418 (2004)
8. Benoit, A., Dufosse, F., Robert, Y.: Filter placement on a pipelined architecture. In: International Symposium on Parallel and Distributed Processing, vol. 0, pp. 1–8 (2009)
9. Bizarro, P., Babu, S., DeWitt, D., Widom, J.: Content-based routing: different plans for different data. In: Proc. of the 31st International Conference on Very Large Data Bases (VLDB), pp. 757–768 (2005)
10. Braga, D., Ceri, S., Daniel, F., Martinenghi, D.: Optimization of multidomain queries on the web. In: Proc. of the VLDB Endowment, vol. 1, pp. 562–573 (2008)
11. Burge, J., Munagala, K., Srivastava, U.: Ordering pipelined query operators with precedence constraints. Technical Report 2005-40, Stanford InfoLab (2005), `http://ilpubs.stanford.edu:8090/705/`
12. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: Niagaracq: a scalable continuous query system for internet databases. In: Proc. of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 379–390 (2000)
13. Condon, A., Despande, A., Hellerstein, L., Wu, N.: Algorithms for distributional and adversarial pipelined filter ordering problems. ACM Transactions on Algorithms 5(2), 24–34 (2009)
14. Deshpande, A., Guestrin, C., Hong, W., Madden, S.: Exploiting correlated attributes in acquisitional query processing. In: Proc. of the 21st International Conference on Data Engineering (ICDE), pp. 143–154 (2005)
15. Deshpande, A., Ives, Z.G., Raman, V.: Adaptive query processing. Foundations and Trends in Databases 1(1), 1–140 (2007)
16. Despande, A., Hellerstein, L.: Flow algorithms for parallel query optimization. In: Proc. of the 24th International Conference on Data Engineering (ICDE), pp. 754–763 (2008)

17. DeWitt, D., Gray, J.: Parallel database systems: the future of high performance database systems. Communications of the ACM 35(6), 85–98 (1992)
18. Epstein, R.S., Stonebraker, M., Wong, E.: Distributed query processing in a relational data base system. In: Lowenthal, E.I., Dale, N.B. (eds.) Proc. of the 1978 ACM SIGMOD International Conference on Management of Data, June 2, pp. 169–180. ACM, New York (1978)
19. Foster, I., Kesselman, C.: The Grid: Blueprint for a New Computing Infrastructure, second edn. Morgan Kaufmann Publishers, San Francisco (2003)
20. Gounaris, A., Smith, J., Paton, N.W., Sakellariou, R., Fernandes, A.A., Watson, P.: Adaptive workload allocation in query processing in autonomous heterogeneous environments. Distrib. Parallel Databases 25(3), 125–164 (2009)
21. Gounaris, A., Yfoulis, C., Sakellariou, R., Dikaiakos, M.D.: Robust runtime optimization of data transfer in queries over web services. In: Proc. of the ACM International Conference on Data Engineering (ICDE), pp. 596–605 (2008)
22. Hellerstein, J.M., Stonebraker, M.: Predicate migration: Optimizing queries with expensive predicates. In: Proc. of the ACM SIGMOD International Conference on Management of Data SIGMOD, pp. 267–276 (1993)
23. Taylor, I., Shields, M., Wang, I.: Resource management of triana p2p services. In: Grid Resource Management (2003)
24. Krishnamurthy, R., Boral, H., Zaniolo, C.: Optimization of nonrecursive queries. In: Proc. of VLDB, pp. 128–137 (1986)
25. Lazaridis, I., Mehrotra, S.: Optimization of multi-version expensive predicates. In: Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 797–808 (2007)
26. Li, J., Deshpande, A., Khuller, S.: Minimizing communication cost in distributed multi-query processing. In: Proc. of the 21st International Conference on Data Engineering (ICDE), pp. 772–783 (2009)
27. Liu, Z., Parthasarathy, S., Ranganathan, A., Yang, H.: Generic flow algorithm for shared filter ordering problems. In: Proc. of the 27th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of database systems (PODS), pp. 79–88 (2008)
28. Liu, Z., Parthasarathy, S., Ranganathan, A., Yang, H.: Near-optimal algorithms for shared filter evaluation in data stream systems. In: Proc. of the ACM International Conference on Management of Data (SIGMOD), pp. 133–146 (2008)
29. Madden, S., Shah, M., Hellerstein, J.M., Raman, V.: Continuously adaptive continuous queries over streams. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 49–60. ACM, New York (2002)
30. Munagala, K., Babu, S., Motwani, R., Widom, J.: The pipelined set cover problem. Technical Report 2003-65, Stanford InfoLab (2003)
31. Munagala, K., Srivastava, U., Widom, J.: Optimization of continuous queries with shared expensive filters. In: Proc. of 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS), pp. 215–224 (2007)
32. Nieto-Santisteban, M.A., Gray, J., Szalay, A.S., Annis, J., Thakar, A.R., O'Mullane, W.: When database systems meet the grid. In: CIDR, pp. 154–161 (2005)
33. Oinn, T., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, M., Carver, T., Glover, K., Pocock, M., Wipat, A., Li, P.: Taverna: a tool for the composition and enactment of bioinformatics workflows. Bioinformatics 20(17), 3045–3054 (2004)
34. Olston, C., Jiang, J., Widom, J.: Adaptive filters for continuous queries over distributed data streams. In: Proc. of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 563–574. ACM, New York (2003)

35. Ozsu, M., Valduriez, P. (eds.): Principles of Distributed Database Systems, 2nd edn. Prentice-Hall, Englewood Cliffs (1999)
36. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-oriented computing: State of the art and research challenges. Computer 40(11), 38–45 (2007)
37. Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., Seltzer, M.: Network-aware operator placement for stream-processing systems. In: Proc. of the 22nd International Conference on Data Engineering (ICDE), pp. 49–60 (2006)
38. Raman, V.: Interactive query processing. PhD thesis, UC Berkeley (2001)
39. Robins, Y., Zelikovski, A.: Improved steiner tree approximation in graphs. In: Proc. of the 11th ACM-SIAM Symposium on Discrete Algorithms, pp. 770–779 (2000)
40. Sabesan, M., Risch, T.: Adaptive parallelization of queries over dependent web service calls. In: Proc. of the International Conference on Data Engineering (ICDE), pp. 1725–1732 (2009)
41. Srivastava, U., Munagala, K., Widom, J.: Operator placement for in-network stream query processing. In: Proc. of the twenty-fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS) (2005)
42. Srivastava, U., Munagala, K., Widom, J., Motwani, R.: Query optimization over web services. In: Proc. of the 32nd Conference on Very Large Databases (VLDB), pp. 355–366 (2006)
43. Terry, D., Goldberg, D., Nichols, D., Oki, B.: Continuous queries over append-only databases. In: Proc. of the 1992 ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 321–330. ACM, New York (1992)
44. Tian, F., DeWitt, D.J.: Tuple routing strategies for distributed eddies. In: Proc. of the 29th Conference on Very Large Databases (VLDB), pp. 333–344 (2003)
45. Tsamoura, E., Gounaris, A., Manolopoulos, Y.: Decentralized execution of linear workflows over web services (submitted for publication)
46. Tsamoura, E., Gounaris, A., Manolopoulos, Y.: Optimal service ordering in decentralized queries over web services. Technical Report, http://delab.csd.auth.gr/~tsamoura/publications.html
47. Yu, J., Buyya, R., Tham, C.K.: Cost-based scheduling of scientific workflow application on utility grids. In: Proc. of the First International Conference on e-Science and Grid Computing (E-SCIENCE), pp. 140–147 (2005)