Optimal Service Ordering in Decentralized Queries over Web Services

Efthymia Tsamoura, Anastasios Gounaris and Yannis Manolopoulos

April 2010

Abstract

The problem of ordering expensive predicates (or filter ordering) has recently received renewed attention due also to emerging computing paradigms such as processing engines for queries over remote Web Services, and cloud and grid computing. The optimization of pipelined plans over services differs from traditional optimization significantly, since execution takes place in parallel and thus the query response time is determined by the slowest node in the plan, which is called the bottleneck node. Although polynomial algorithms have been proposed for several variants of optimization problems in this setting, the fact that communication links are typically heterogeneous in wide-area environments has been largely overlooked. Our proposal is the first attempt, to the best of our knowledge, that tries to optimize linear orderings of services when the services communicate directly with each other and the communication links are heterogeneous. We propose a novel optimal algorithm to solve this problem efficiently, which is thoroughly evaluated through detailed experiments.

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 enterprises and organizations. This trend has altered, to an extent, the way complex tasks are formulated, giving rise to approaches that rely on composition of services to be executed in a parallel and distributed manner (e.g., [19, 1, 22]). As a consequence, there is a growing interest in systems that are capable of processing complex tasks formulated as Web Service (WS) workflows utilizing remote computational resources. In [24], the notion of Web Service Management System (WSMS) is introduced as a general purpose system, which possesses such capabilities.

In a WSMS, processing of data takes place through (remote) calls to WSs. The latter provide an interface of the form $WS : \mathcal{X} \rightarrow \mathcal{Y}$, where $\mathcal{X}$ and $\mathcal{Y}$ are sets of attributes, i.e. given values for attributes in $\mathcal{X}$, WS returns values for the attributes in $\mathcal{Y}$, as shown in the following example adapted from [24]. In the generic case, the input tuples may have more attributes than $\mathcal{X}$, while attributes in $\mathcal{Y}$ 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.

In many cases, the problem of optimizing queries over WSs becomes equivalent to that of an optimal ordering of a query's WS calls. Usually, the goal of the optimization is to minimize the response time of queries, even though there may be other metrics of interest, such as total time, monetary cost and aggregate resource utilization. In this work, we focus on the minimization of query response time, exclusively.

In one aspect, the problem of optimal ordering of WS with a view to minimizing the response time may resemble the problem of ordering commutative filters in pipelined queries with conjunctive predicates [15, 11], 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 [3]. However, there are also many substantial differences given that there may exist precedence constraints between the WSs, selectivities may be higher than 1 (e.g., WS_3 in the example) and, typically, the execution of queries over WSs takes place in a both distributed and parallel manner.

More specifically, each WS is executed on a different node and the results of one WS may immediately be passed on to the next service in a pipelined fashion, so that the tuples already processed by a WS are processed by the subsequent WS in the plan at the same time as the former processes new input tuples.

According to the pipelined execution model, when no tuples are dropped or new tuples are generated, the maximum rate at which input tuples can be processed through a single plan equals the minimum processing rate of all services; the corresponding service, i.e., the WS that spends, on average, the most time per input tuple, is termed the bottleneck WS. This model imposes new optimization challenges. For example, the query response time is no longer the sum of the cost in time units of all the WSs in the pipelined plan, but is determined by the slowest node [8, 5, 24].

The problem of minimizing the bottleneck cost has received significant attention recently. In [24], along with the introduction of WSMSs, an efficient optimization algorithm is presented that considers precedence constraints among the WSs. Another characteristic of this work is that it can deal with any selectivity values and build plans where the output of a service is fed to multiple services simultaneously. The proposals in [8, 5] introduce faster algorithms that maximize the data flow by defining the set of interleaving plans along with the proportion of tuples routed to each plan in order to maximize the aggregate processing rate; nevertheless, all the plans are linear, i.e., each WS has at most one input service and one output. More detailed discussion of the related work is deferred to Section 5. However, a common feature of all these algorithms is that they do not take the potentially heterogeneous communication links between the services into account, which is significant when the execution is decentralized given also that the communication cost may be the dominant cost. This is in line with the WSMS in [24], which assumes that the output of a service is fed to the subsequent service indirectly through a central management component thus annihilating the need to consider the different communication costs explicitly. As such, in existing proposals, the bottleneck cost depends solely on the service costs and selectivities, instead of taking also into account the inter-service communication cost, especially when the execution occurs in a wide are environment. In the

previous example, the optimal ordering may differ when all services are at a single place, when they are all at a different place, and when, for instance, only WS_1 and WS_5 are co-located.

The main contribution of this work is that it addresses the afore-mentioned limitation by proposing an efficient algorithm for the single optimal ordering of services when the services communicate directly with each other and the communication costs between the services may differ. Our algorithm is based upon the branch and bound optimization paradigm and operates regardless of any precedence constraints. It can be deemed as an extension to [24] accounting for decentralized execution, except that we are only interested in linear orderings. The thorough evaluation of the proposal shows that it is efficient and can result in significant reductions of the response time up to an order of magnitude in many realistic scenarios.

The remainder of this paper is structured as follows. The problem we deal with is formally introduced in Section 2. Section 3 presents the algorithm in detail discussing its main concept, correctness proofs, and implementation issues. The evaluation results appear in Section 4. Section 5 discusses the related work, and Section 6 concludes the paper.

2 Problem formulation

In our parallel execution model, each WS runs on a different node in a separate thread that processes input tuples and sends output tuples to the next service sequentially; our solution however can be applied to the case when separate threads are responsible for data processing and transmission in a straight-forward manner. Let c_i be the average time needed by WS_i to process an input tuple (also referred to as the cost of WS_i), σ_i the selectivity of WS_i and $t_{i,j}$ the time needed to transfer a tuple from WS_i to WS_j[1]. We assume that c_i, σ_i and $t_{i,j}$ are constants and independent of the input attribute values. We further assume that the selectivities of WSs are independent of each other, and, in the

[1]In practice, tuples are transmitted in blocks [10, 24]; $t_{i,j}$ is the cost to transmit a block divided by the number of tuples it contains.

generic case, can be greater than 1. A constrained service WS_i has at least one prerequisite service WS_j, which is denoted as $WS_j \prec WS_i$. The precedence constraints are part of the input (e.g, in the form of a DAG).

A plan $\mathcal{S}$ consists of a linear ordering of the WSs, which respects any precedence constraints and its response time is given by the bottleneck cost metric, in accordance to [24]:

$$cost(\mathcal{S}) = \max_{i|WS_i \in \mathcal{S}} \left\{ \prod_{k|WS_k \in P_i(\mathcal{S})} \sigma_k(c_i + \sigma_i t_{i,i+1}) \right\} \quad (1)$$

where $P_i(\mathcal{S})$ is the set of WSs that are invoked before WS_i in the plan $\mathcal{S}$. Throughout the paper we will refer to $T_{i,j} = c_i + t_{i,j}\sigma_i$ as the aggregate cost of WS_i with respect to WS_j. If $t_{i,j}$ is equal for all service pairs, the problem can be solved in polynomial time, as shown in [24]. Here we deal with the generic –and more realistic– case, where $t_{i,j}$ may differ, for which to the best of out knowledge, there is no polynomial solution. More specifically, the problem we deal with in this work is formulated as follows:

Problem Formulation 1 *Given a set W of N WSs $W = \{WS_0, WS_1, \ldots, WS_{N-1}\}$, where each one of them is allocated on a host machine, find the linear plan $\mathcal{S}$, which minimizes the bottleneck cost metric given by Eq. (1)*[2].

3 Optimal linear plan construction algorithm

3.1 Our approach in brief

The proposed algorithm is based on the branch-and-bound optimization approach. Let $\mathcal{C} = WS_{k(0)}WS_{k(1)} \ldots WS_{k(n)}$ be a partial linear plan. k is used throughout the paper to denote the mapping from positions in the WSs to the indices of the WSs at those positions.

Two cost metrics guide the plan building process, ϵ and $\bar{\epsilon}$ respectively. The former corresponds to the bottleneck cost of $\mathcal{C}$, while the latter is the maximum possible cost

[2] $t_{N-1,N}$ is always set to 0.

that may be incurred by WSs not currently included in $\mathcal{C}$. Cost $\overline{\epsilon}$ is utilized to speed up the algorithm. For simplicity, we will first discuss the case where the selectivities are not greater than 1. ϵ and $\overline{\epsilon}$ are given by the following equations:

$$\epsilon = \max\left\{T_{k(0),k(1)}, \max_{1\leq i<n}\left\{\left(\prod_{j=0}^{i-1}\sigma_{k(j)}\right)T_{k(i),k(i+1)}\right\}\right\}, \tag{2}$$

$$\overline{\epsilon} = \max_{l,r}\left\{\begin{array}{ll}\left(\prod_{j=0}^{n}\sigma_{k(j)}\right)T_{l,r}, & WS_l \notin \mathcal{C} \\ \left(\prod_{j=0}^{n-1}\sigma_{k(j)}\right)T_{l,r}, & WS_l = WS_{k(n)}\end{array}\right\} \tag{3}$$

$WS_r \notin \mathcal{C}$ in both cases.

The algorithm proceeds in two phases, namely the *expansion* and the *pruning* one. During expansion, new WSs are appended to a partial plan $\mathcal{C}$, while during the latter phase WSs are pruned from $\mathcal{C}$ with a view to exploring additional orderings. If, for a partial plan $\mathcal{C}$, the condition $\epsilon < \overline{\epsilon}$ is met, this means that the bottleneck cost of the plan beginning with $\mathcal{C}$ depends on the ordering of the services not yet included; so a new WS_r is appended to $\mathcal{C}$. On the other hand, if condition $\epsilon \geq \overline{\epsilon}$ is met, then the order in which the rest WSs may be appended to $\mathcal{C}$ does not affect its bottleneck cost ϵ, since the maximum possible cost $\overline{\epsilon}$ that may be incurred cannot be higher than ϵ. In that case, the linear plan $\mathcal{C}$ is essentially a solution and the pruning step is triggered. Partial plans are also pruned when they cannot form a prefix of an optimal solution. Let $\mathcal{C} = WS_{k(0)}WS_{k(1)}\ldots WS_{k(n)}$, where $0 \leq n < N$ and $WS_{k(i)}$ be the bottleneck WS of $\mathcal{C}$, where $0 \leq i \leq n$. Then $\mathcal{C}$ is pruned as follows:

$$\mathcal{C} = \left\{\begin{array}{ll}\emptyset, & i = 0, \\ WS_0WS_1\ldots WS_{i-1}, & 0 < i \leq n < N\end{array}\right. \tag{4}$$

To further improve the efficiency of the algorithm, the prefixes up to the bottleneck service $WS_{k(i)}$ (denoted as $\mathcal{C}'$) of the plans for which the expansion phase has been completed are stored in a list $\mathcal{V}$. To avoid investigating the same solutions multiple times, the function $\pi(\mathcal{X})$ is employed, where $\mathcal{X}$ is a (potentially partial) plan. This function returns $true$ if no WS plan stored in $\mathcal{V}$ is prefix of $\mathcal{X}$. As will be discussed later, the

plans considered must satisfy the $\pi()$ function thus yielding better running times without compromising the optimality of the algorithm.

3.2 Detailed Algorithm Description

The complete algorithm is shown in Fig. 1, where $\mathcal{S}$ denotes the best linear plan found so far and ρ its bottleneck cost. In every iteration of the algorithm, the cost metrics ϵ and $\overline{\epsilon}$ are evaluated. $F(\mathcal{C})$ is the set of all WSs for which all the prerequisite WSs have already been added to $\mathcal{C}$. More formally $F(\mathcal{C})$ is given by:

$$F(\mathcal{C}) = \{WS_i | WS_i \notin \mathcal{C} \wedge M_i \subseteq \mathcal{C}\} \tag{5}$$

and M_i is the set of all WSs that must appear before WS_i, i.e., $M_i = \{WS_j | WS_j \prec WS_i\}$. Obviously, for unconstrained services, $M_i = \emptyset$.

There are three cases depending on the values of ϵ, $\overline{\epsilon}$ and ρ in a partial plan $\mathcal{C}$:

- $\epsilon < \overline{\epsilon},\ \epsilon < \rho$ (lines 12-24): if $\mathcal{C}$ is empty, e.g., it is the initial iteration, the algorithm searches for the most promising WS pair that has not been considered yet. Such a WS pair must satisfy the following equation

  $$T_{l,r} = \min_{i,j | M_i = \emptyset \wedge M_j \subseteq \{WS_i\} \wedge \pi(WS_i WS_j) = true} \{T_{ij}\} \tag{6}$$

  If the partial plan is not empty, the algorithm searches for a new WS WS_r, such that:

  $$T_{k(n),r} = \min_{WS_j \in F(\mathcal{C}) \wedge \pi(\mathcal{C} WS_j) = \text{true}} \{T_{k(n),j}\}, \tag{7}$$

  These subcases comprise the expansion case. In case where no WS_r can be found, the last WS of $\mathcal{C}$ is pruned (lines 18-20) to support cases in which no service can be appended, e.g., all plans with prefix $\mathcal{C}$ have been examined. Note that the condition $\epsilon < \rho$ is considered along with $\epsilon < \overline{\epsilon}$, because it is worth performing the expansion phase only for plans with $\epsilon < \rho$.

- $\overline{\epsilon} \leq \epsilon < \rho$ (lines 25-30): the current best linear plan $\mathcal{S}$ is set to $\mathcal{C}$ and its bottleneck cost ρ is updated. Condition $\epsilon \geq \overline{\epsilon}$ ensures that the bottleneck cost of $\mathcal{C}$, which

Inputs

N: Number of input WSs.

T: An [N x N] matrix, where $T_{i,j} = c_i + t_{i,j}\sigma_i$.

Outputs

$\mathcal{S}$: optimal linear plan

$\mathcal{S} \leftarrow \emptyset$; {$\mathcal{S}$ is the current best linear plan}
$\rho \leftarrow \infty$; {ρ is the bottleneck cost of $\mathcal{S}$}
$\mathcal{V} \leftarrow \emptyset$; {$\mathcal{V}$ is a list of partial linear plans.}
$\mathcal{C} \leftarrow \emptyset$;
$F(\mathcal{C}) \leftarrow \{WS_i | M_i = \emptyset\}$;
while true **do**
  Estimate ϵ using Eq. (2);
  Estimate $\overline{\epsilon}$ using Eq. (3);
  if $\epsilon \geq \rho$ **then**
    $\mathcal{V}.push(\mathcal{C}')$;
    Trim $\mathcal{C}$ following Eq. (4);
  else if $\epsilon < \overline{\epsilon} \wedge \epsilon < \rho$ **then**
    if $\mathcal{C} = \emptyset$ **then**
      Find services WS_l and WS_r using Eq. (6);
      $\mathcal{C} = WS_l WS_r$;
    else if $\mathcal{C} \neq \emptyset$ **then**
      Find a WS_r using Eq. (7);
      if no such WS can be found **then**
        Trim $\mathcal{C}$ using Eq. (4), where the bottleneck WS is set to $WS_{k(n)}$;
        $\mathcal{V}.push(\mathcal{C}')$;
      else
        Append the WS_r to $\mathcal{C}$;
      end if
    end if
  else if $\overline{\epsilon} \leq \epsilon < \rho$ **then**
    $\mathcal{V}.push(\mathcal{C}')$;
    $\mathcal{S} \leftarrow \mathcal{C}$;
    Trim $\mathcal{C}$ using Eq. (4);
    $\rho \leftarrow \epsilon$;
  end if
  Update $F(\mathcal{C})$ using Eq. (5);
  if Termination Condition 1 is true **then**
    return $\mathcal{S}$;
  end if
end while

Figure 1: The proposed algorithm.

i \ j	1	2	3	4	5	6	7	8	9	10
1	-	10.43	21.89	13.80	29.01	15.35	23.56	21.65	14.63	20.30
2	16.58	-	28.82	21.28	34.15	33.30	42.69	33.97	24.19	32.30
3	34.88	31.96	-	23.21	32.33	32.06	32.51	29.58	42.60	15.23
4	20.52	20.86	20.87	-	31.48	34.93	33.71	32.58	28.62	33.32
5	22.47	21.05	19.94	20.73	-	20.97	19.52	19.31	19.45	17.37
6	18.19	23.51	21.26	25.09	23.76	-	20.50	21.93	22.73	27.31
7	39.60	50.49	34.66	41.50	32.27	32.47	-	31.38	40.49	42.13
8	24.33	27.74	19.41	27.62	18.86	23.41	19.10	-	30.95	13.22
9	14.32	15.46	22.59	18.27	16.89	19.39	20.66	21.96	-	18.97
10	27.08	30.48	10.34	33.14	13.90	39.98	34.23	18.22	28.98	-

Table 1: Aggregate cost matrix $\mathbf{T}$.

is lower than the current lowest cost, will not increase if new WSs are appended. After that, $\mathcal{C}$ is pruned following the Eq. (4) and the algorithm continues. The intuition behind Eq. (4) is as follows. $WS_{k(i+1)}$ in Eq. (4) satisfies either Eq. (6), or (7), i.e. it is the WS such that $WS_{k(i)}$ has the minimum cost $T_{k(i),k(i+1)}$. Thus, the cost that may be incurred by any other WS appended to $WS_{k(i)}$, will be higher than the current bottleneck cost, i.e., it is worthless to investigate plans with prefix $WS_{k(0)}...WS_{k(i)}$.

- $\epsilon \geq \rho$ (lines 9-11): $\mathcal{C}$ cannot yield an optimal solution, since its bottleneck cost is higher than the bottleneck cost of $\mathcal{S}$. Thus, $\mathcal{C}$ is pruned following Eq. (4).

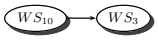
The algorithm can safely terminate when the less expensive pair of WSs satisfying the π function cannot improve the current bottleneck cost, which means that the best possible linear plan not yet visited has at least as high bottleneck cost ϵ as ρ.

Termination Condition 1 *Let ρ be the minimum bottleneck cost found so far. The algorithm terminates, when there are no two services WS_l and WS_r such that:*

$$T_{l,r} \geq \rho,\ M_l = \emptyset \wedge M_r \subseteq \{WS_l\} \wedge \pi(WS_l WS_r) = \text{true} \tag{8}$$

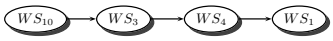
The proof of the algorithm's correctness is in a subsequent section.

1st iteration (at the beginning $\epsilon = 0, \overline{\epsilon} = 50.4958, \rho = \infty$)



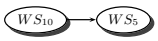
2nd iteration (at the beginning $\epsilon = 10.3425, \overline{\epsilon} = 41.5973, \rho = \infty$)

3rd iteration (at the beginning $\epsilon = 20.7436, \overline{\epsilon} = 30.7074, \rho = \infty$)



4th iteration (at the beginning $\epsilon = 20.7436, \overline{\epsilon} = 18.8983, \rho = \infty$)

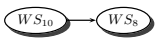
5th iteration (at the beginning $\epsilon = 0, \overline{\epsilon} = 45.1255, \rho = 20.7436$)



6th iteration (at the beginning $\epsilon = 13.9069, \overline{\epsilon} = 20.0812, \rho = 20.7436$)

7th iteration (at the beginning $\epsilon = 17.2575, \overline{\epsilon} = 7.29314, \rho = 20.7436$)

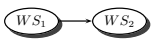
8th iteration (at the beginning $\epsilon = 0, \overline{\epsilon} = 45.1255, \rho = 17.2575$)



9th iteration (at the beginning $\epsilon = 18.2236, \overline{\epsilon} = 41.3758, \rho = 17.2575$)

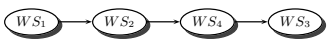
$\mathcal{C} = \emptyset$

10th iteration (at the beginning $\epsilon = 0, \overline{\epsilon} = 50.4958, \rho = 17.2575$)



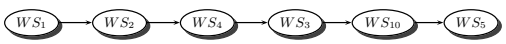
11th iteration(at the beginning $\epsilon = 10.4372, \overline{\epsilon} = 26.2738, \rho = 17.2575$)

12th iteration(at the beginning $\epsilon = 13.0996, \overline{\epsilon} = 17.0269, \rho = 17.2575$)



13th iteration (at the beginning $\epsilon = 13.0996, \overline{\epsilon} = 15.33, \rho = 17.2575$)

14th iteration (at the beginning $\epsilon = 13.0996, \overline{\epsilon} = 13.2628, \rho = 17.2575$)



15th iteration (at the beginning $\epsilon = 13.0996, \overline{\epsilon} = 6.21801, \rho = 17.2575$)

Figure 2: An example of the proposed algorithm.

WS_i	1	2	3	4	5	6	7	8	9	10
σ_i	0.61	0.79	0.92	0.73	0.17	0.40	0.93	0.91	0.41	0.89

Table 2: Selectivities of WSs in $\mathcal{W}$.

3.3 An example

In the following, an example for building an optimal plan for unconstrained WSs employing the proposed algorithm is presented. Let $\mathcal{W} = \{WS_1, \ldots, WS_{10}\}$ be a set of 10 services with corresponding aggregate costs and selectivities shown in Table 1 and 2, respectively. The values in the tables follow a gaussian distribution. Fig. 2 shows the partial plans at the end of each iteration.

Initially, $\epsilon = 0 < \overline{\epsilon} = T_{7,2} = 50.4958$, $\epsilon < \rho = \infty$, and $\mathcal{C} = \emptyset$. The algorithm starts by identifying the WS pair, which incurs the minimum cost, see lines 13-15 of Fig. 1. The corresponding WSs are $WS_{10}WS_3$. After that, $\mathcal{C} = WS_{10}WS_3$. In the second iteration, since $\epsilon = 10.3425 < \overline{\epsilon} = \sigma_{10} \times \sigma_3 \times T_{7,2} = 41.5973$, $\epsilon < \rho = \infty$, and $\mathcal{C} \neq \emptyset$, a new WS is appended to $\mathcal{C}$, which must satisfy the Eq. (7); that service is WS_4. In the third iteration, since $\epsilon = 20.7436 < \overline{\epsilon} = \sigma_{10} \times \sigma_3 \times \sigma_4 \times T_{7,2} = 30.7074$, $\epsilon < \rho = \infty$, and $\mathcal{C} = \emptyset$, the service WS_1 is appended to $\mathcal{C}$ forming the partial plan $\mathcal{C} = WS_{10}WS_3WS_4WS_1$. Now, since $\epsilon = 20.7436 > \overline{\epsilon} = \sigma_{10} \times \sigma_3 \times \sigma_4 \times \sigma_1 \times T_{7,2} = 18.8983$ and $\epsilon < \rho = \infty$, a solution is found. According to steps in lines 25-30 of Fig. 1, $\mathcal{S}$ is set to $\mathcal{C}$, $\rho = 20.7436$ and $\mathcal{C}$ is pruned following the Eq.(4). After the pruning $\mathcal{C} = WS_{10}$ (the bottleneck WS is WS_3). The Termination Condition 1 is not triggered given that there exists a two service prefix that satisfies π and its cost is lower than ρ: $T_{1,2} = 10.43$.

In the fifth iteration, since $\epsilon = 0 < \overline{\epsilon} = 45.1255$, $\epsilon = 0 < \rho = 20.7436$, and $\mathcal{C} \neq \emptyset$, a new WS is appended to $\mathcal{C} = WS_{10}$; that is WS_5. A new WS is also appended in the sixth iteration forming the partial plan $\mathcal{C} = WS_{10}WS_5WS_8$. In the seventh iteration, since $\epsilon = 17.2575 > \overline{\epsilon} = 7.29314$ and $\epsilon < \rho = 20.7436$, another solution is found: $\mathcal{S} = WS_{10}WS_5WS_8$, and its bottleneck cost updates ρ; subsequently, $\mathcal{C}$ is pruned following Eq.(4) so that $\mathcal{C} = WS_{10}$. In the above iterations, the Termination Condition 1 is not triggered because the cost $T_{1,2}$ is lower than the ρ values. In the eight iteration, WS_8 is appended to $\mathcal{C} = WS_{10}$, since it satisfies the Eq.(7). However, in the ninth iteration, the

partial plan is set to $\mathcal{C} = \emptyset$, as $\epsilon = 18.2236 > \rho = 17.2575$ and the bottleneck WS is the first one, i.e., WS_{10}. As a result, any plan starting with WS_{10} no longer satisfies the π function.

In the tenth iteration, the WSs which satisfy the Eq.(6) form the partial plan $\mathcal{C}$, since $\epsilon = 0 < \overline{\epsilon} = T_{7,2} = 50.4958$, $\epsilon < \rho = 17.2575$, and $\mathcal{C} = \emptyset$. The result is the ordering $\mathcal{C} = WS_1WS_2$. In the iterations 11-14 new WSs are appended to $\mathcal{C}$, forming the partial plan $\mathcal{C} = WS_1WS_2WS_4WS_3WS_{10}WS_5$. In the fifteenth iteration, a new solution is found, since $\epsilon = 13.0996 > \overline{\epsilon} = 6.21801$ and $\epsilon < \rho = 17.2575$. Following the steps in lines 25-30, $\mathcal{S} = WS_1WS_2WS_4WS_3WS_{10}WS_5$, the bottleneck service is WS_2 and ρ is set to 13.0996. After the pruning $\mathcal{C} = WS_1$. However, the Termination Condition 1 is now triggered, since the cost of the less expensive WS pair satisfying π, which is WS_8WS_{10}, is now higher than ρ: $T_{8,10} = 13.2293 > \rho = 13.0996$. So the algorithm terminates, after having essentially explored all the $10!$ orderings in just 15 iterations.

3.4 Proof of Correctness

Lemma 1 *If ϵ is the bottleneck cost of $\mathcal{C}$, any plan with prefix $\mathcal{C}$ cannot have a lower bottleneck cost.*

This non-decreasing property of ϵ with regards to the size of the partial plan derives directly from Eq. (1). Also, as explained in the previous subsections, the following lemma holds:

Lemma 2 *If for a partial plan $\mathcal{C}$, $\epsilon \geq \overline{\epsilon}$, then, any plan with prefix $\mathcal{C}$ has cost ϵ.*

This derives from Eq. (2) and (3) and the fact that selectivities are not greater than 1.

Lemma 3 *No plan $\mathcal{C}$ with prefix any of the plans stored in $\mathcal{V}$ can have bottleneck cost $\epsilon < \rho$, where ρ is the minimum bottleneck cost found so far.*

The plans in $\mathcal{V}$ fall into two categories. Firstly, $\mathcal{V}$ includes prefixes of partial plans up to and including their bottleneck service; the cost of at least one of such bottleneck services is ρ while the costs of other bottlenecks are higher. However, for any plan used to

construct $\mathcal{V}$, the service appended to the plan just after the bottleneck service during the expansion phase must have satisfied either Eq. (6), or (7), resulting in bottleneck cost $\epsilon \geq \rho$. Because of Eq. (6) and (7), any plan $\mathcal{C}$ produced by appending a service to $\mathcal{V}$ has cost $\epsilon \geq \rho$, too. So, with the help of the first lemma, this lemma holds as well. Secondly, $\mathcal{V}$ includes partial plans for which no new services can be appended, due to either precedence constraints or the fact that all combinations have been already explored (line 20 of Fig. 1). The former subcase cannot lead to any solutions, whereas the latter is similar to the first case. This completes the proof of the lemma, and also shows the correctness of using the $\pi()$ function, which builds upon this lemma.

Theorem 1 *The algorithm finds the optimal solution.*

A sketch of the proof is as follows. In order to prove the correctness of the algorithm, we must prove that all possible orderings have been checked, either directly or indirectly, the termination condition is correct and the output is a valid solution. The algorithm, with the help of the $\pi()$ function does not explore plans the prefix of which is stored $\mathcal{V}$. We have proven that this does not compromise its optimality. The algorithm exits when any WS pair that is a valid beginning of a plan does not have a cost lower than the currently lowest bottleneck cost. In general, there are $n!$ orderings, where n is the number of services; of course this can be reduced due to precedence constraints. However, there are at most $n(n-1)$ prefixes of size two. If all these prefixes satisfy the termination condition, with the help of the first lemma we can show that all $n!$ orderings cannot have a bottleneck cost that is lower than ρ. Finally, $\mathcal{S}$ in Fig. 1 is a partial plan, but, as the second lemma shows, any plan with prefix $\mathcal{S}$ can form a complete optimal ordering.

Note that the termination condition can be reduced to simpler statements in some specific cases. For example, when there are precedence constraints such that only a single WS can be at position 0, e.g., it is the service responsible for source data generation, then the termination condition can be reduced to the statement that the algorithm can exit if the bottleneck position reaches the first position $WS_{k(0)}$.

3.5 Proliferative Services

Thus far, we have discussed the case when service selectivities are not higher than 1. If there exist $\sigma_i > 1$, then the same algorithm is still valid; the only change is in the way $\overline{\epsilon}$ is computed in Eq. (3). More specifically, $\overline{\epsilon}$ in Eq. (3) is multiplied by the product of all $\sigma_i > 1$ such that $WS_i \notin \mathcal{C}$. The proof of correctness is similar to the case when WSs are selective.

3.6 Implementation issues

In the current subsection, we will discuss some implementations issues. We have employed some simple data structures in order to speed up the execution of expensive operations, namely the identification of the next WS to be appended to a partial plan (Eq. (7)), the evaluation of $\overline{\epsilon}$ and the check of the termination condition. Note that the computation of the bottleneck cost ϵ of $\mathcal{C}$ and the detection of the bottleneck service require linear time; also $\pi()$ can be efficiently implemented with the help of a prefix tree.

To speed up the identification of the next WS, a preprocessing step takes place before the algorithm execution. According to this step, for each service WS_i a doubly-linked list L_i of all services WS_j in increasing order of the corresponding aggregate cost $T_{i,j}$ is constructed. Thus, the next WS to be added after $WS_{k(n)}$ is found using the following simple search approach: we start from the head of list $L_{k(n)}$. If the current WS is not included in $\mathcal{C}$ and all its prerequisites services are included in $\mathcal{C}$, then the desirable WS is found, otherwise the search continues. The computation of $\overline{\epsilon}$ is performed using an analogous approach. For every WS_i that it is either the last WS of $\mathcal{C}$ or does not belong to $\mathcal{C}$, the search starts from the end of L_i, since the maximum possible process/transfer cost must be found. The search in every L_i stops when the first WS not currently included in $\mathcal{C}$ is found and the maximum aggregate cost among those found in every L_i is returned.

Finally, the check of the termination condition can be efficiently done with the help of a min heap. The contents of this heap vary depending on whether the WSs are constrained or not. In the former case, the min heap contains all WS couples (WS_iWS_j), while in the latter case, it contains only the couples that can constitute a valid prefix of a plan.

The values of the heap are the costs $T_{i,j}$ of the corresponding couples. Every time we check whether the termination condition is met, the root of the min heap is accessed and the plan $\mathcal{X} = WS_iWS_j$ is formed from the WS couple (WS_iWS_j) stored in the root of the min heap. If the partial plan $\mathcal{X}$ satisfies the $\pi(\mathcal{X})$, then the corresponding cost is kept. Otherwise the root of the min heap is deleted and the next new root element is accessed. From the above, it follows that the complexity of this operation is $O(\log k)$, where $k \leq N^2$, while the complexity of naive approach is $O(N^2)$.

4 Evaluation

4.1 Experimental Setting

In the current section, we experimentally evaluate the proposed algorithm, which will be referred to as Optimal Linear Plan Constructor (OLPC). The evaluation is conducted to investigate firstly the algorithm's performance, and secondly, the algorithm's efficiency. The performance of the algorithm is evaluated through the comparison of the response times of a wide range of query plans produced by OLPC and the *Greedy* algorithm in [24]. The efficiency is measured in terms of the absolute time needed to construct the plans and of the number of iterations in OLPC, when implemented exactly as shown in Fig. 1.

The simulation environment is defined by a five-dimensional vector $A = [\mathbf{G_c}(\overline{c}, \sigma_c)$, $\mathbf{G_t}(\lambda\overline{c}, \gamma\lambda\overline{c}), \mathbf{U}_\sigma(l, u), p, N]$. $\mathbf{G_c}$ corresponds to the distribution of the processing cost values (c_i) of the input services. More specifically, throughout the evaluation, we assume that the processing costs of WSs follow a gaussian distribution with mean value $\overline{c}$ and standard deviation σ_c. The data transmission costs $t_{i,j}$ follow a gaussian distribution, too (denoted as $\mathbf{G_t}(\lambda\overline{c}, \gamma\lambda\overline{c})$). The mean value of $\mathbf{G_t}$ is related to $\overline{c}$, in order to enable the investigation of the impact of the ratio of computation cost to the communication cost on the performance. More precisely, the mean value of $t_{i,j}$, $\overline{t}$ is given by $\overline{t} = \lambda\overline{c}$ where $\lambda \in \{0.25, 0.5, 0.75, 1, 1.25, \ldots, 10\}$. Its standard deviation is given by $\sigma_t = \gamma\overline{t}$ where $\gamma \in \{0.1, 0.2, 0.3, \ldots, 0.9\}$.

$\mathbf{U}_\sigma(l, u)$ represents the uniform distribution of the services' selectivity values. l and u are the corresponding lower and the upper bounds, respectively. Finally, p and N are single-valued parameters. The former is used to create precedence constraints and corresponds to the probability for one service WS_i to have a random $WS_j, i < j$ as its prerequisite; for unconstrained WSs, $p = 0$. N is the number of input WSs; we have experimented with values of $N \in \{10, 20, 30, \ldots, 250\}$. An example of a simulation environment is $[\mathbf{G_c}(10, 5), \mathbf{G_t}(20, 2), \mathbf{U}_\sigma(0, 1), 0, 10]$, which corresponds to the case where there are 10 unconstrained services, with selectivities uniformly distributed between 0 and 1. The processing costs of these WSs follow a gaussian distribution with mean value $\overline{c} = 10$ and standard deviation $\sigma_c = 5$. The communication costs follow a gaussian distribution with mean value $\overline{t} = 20$ (i.e., $\lambda = 2$ and standard deviation $\sigma_t = 2$ (i.e., $\gamma = 0.1$).

Each simulation instance is a random realization of the simulation environment, so that five-dimensional vectors of the form $[\mathbf{c}, \mathbf{t}, \boldsymbol{\sigma}, p, N]$ are generated, where $\mathbf{c}, \mathbf{t}$ and $\boldsymbol{\sigma}$ are matrices populated according to $\mathbf{G_c}(\overline{c}, \sigma_c), \mathbf{G_t}(\lambda\overline{c}, \gamma\lambda\overline{c}), \mathbf{U}_\sigma(l, u)$, respectively.

Finally, ρ_A and ρ'_A denote the response times of the plan produced by OLPC and Greedy respectively, when they are executed on the same simulation instance of A. ζ_A is the number of iterations that OLPC performs. In the experiments, we fix $\overline{c}$ to 10 and σ_c to 5 and we vary only the λ and γ variables in order to generate settings with a wide range of different ratios between mean processing and communication costs. Note that we have chosen a not very low standard deviation value, since we want our environment to better simulate heterogeneous, wide area settings. Also, we do not compare plans with service selectivities higher than 1, because, for such services, the Greedy algorithm builds parallel plans, whereas OLPC builds only linear plans [24]; extending OLPC to support parallel plans is left for future work.

The general remarks of the evaluation can be summarized in the following lines: OLPC outperforms Greedy by several factors. In many realistic scenarios the performance improvements are of an order of magnitude, whereas we have noticed improvements by a factor as high as 164 (Table 3). Furthermore, the algorithm is very efficient, as it requires negligible running time and a limited number of iterations.

4.2 No precedence constraints

We first investigate scenarios with no precedence constraints. Since the selectivity of each WS is between 0 and 1, the plans that Greedy algorithm produces are linear orderings of the WSs by increasing processing time ignoring selectivities. The following experiments are conducted in order to study the convergence between the response times of plans produced by Greedy and the response times of OLPC plans. The response time is given by Eq. (1). OLPC always produces the optimal serial WS plan. As such, all the experiments that follow deal with the extent to which the response times of the Greedy plans are higher than the response times of the OLPC plans, for a given simulation instance. Intuitively, the performance is affected by various parameters. In our work we concentrate on four of them, namely the ratio between processing and transferring costs of tuples, the network heterogeneity, the WS selectivities and the number of input services.

Experiment I: impact of the number of WSs on performance. Here we compare the performance of OLPC and Greedy when the number of input WSs varies. This is done with the help of the following response time ratio metric

$$r(\lambda, N) = \frac{\rho'_{[\mathbf{G_c}(10,5),\mathbf{G_t}(10\lambda,10\lambda\gamma),\mathbf{U}_\sigma(0,1),0,N]}}{\rho_{[\mathbf{G_c}(10,5),\mathbf{G_t}(10\lambda,10\lambda\gamma),\mathbf{U}_\sigma(0,1),0,N]}}$$

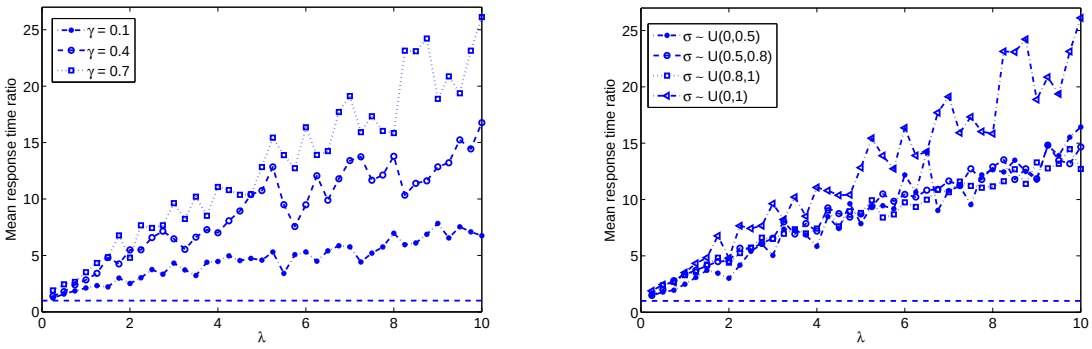
The experimental results for different values of N and λ are shown in Table 3. When $r(\lambda, N) = 1$, a Greedy plan has exactly the same response time as a plan built by OLPC for the same simulation instance. In general, we observe that (i) OLPC can yield significant performance improvements of several factors (up to 164); and (ii) for fixed λ, γ and σ values, the response times of the plans built by Greedy ρ' tend to increasingly deflect from the response times of the OLPC plans ρ as the number of the input services increases. For example, in the lower part of Table 3, for $N = 250$ WSs, the maximum response time ratio is 64.13 times, while for $N = 10$ WSs it is 13.32 times.

Experiment II: impact of heterogeneity on performance. Now, we turn our attention to two other parameters, namely the ratio between the processing and the transferring costs of tuples and the network heterogeneity. A closer look to Table 3 shows that for fixed values of γ and N, the response time deviations between Greedy and OLPC plans

$\lambda \backslash N$	10	40	70	100	130	160	190	220	250
$\gamma = 0.1$									
0.5	1.00	1.07	1.01	1.03	2.20	1.02	1.61	4.64	1.08
1.5	1.79	1.72	1.10	1.40	7.05	2.98	2.35	3.08	1.20
2.5	2.66	1.21	2.00	4.22	1.21	3.99	2.65	13.88	20.13
3.5	1.16	6.09	1.54	2.38	1.28	2.47	1.33	2.80	3.90
4.5	1.17	3.71	1.00	3.13	6.48	11.29	9.18	14.97	4.57
5.5	2.65	1.65	2.12	2.67	1.27	4.72	1.94	3.09	1.19
6.5	1.20	1.96	6.36	6.76	4.08	5.01	1.63	29.49	13.18
7.5	3.98	6.76	3.98	1.38	5.71	11.99	2.79	4.92	9.27
8.5	3.33	1.23	6.55	10.65	10.50	1.48	4.39	20.40	14.68
9.5	5.42	10.36	4.70	11.69	8.12	4.58	6.98	14.75	2.00
$\gamma = 0.4$									
0.5	1.21	1.31	4.27	1.35	1.23	1.41	1.36	3.21	5.24
1.5	2.23	1.39	3.43	2.64	3.11	3.58	1.69	22.25	5.22
2.5	1.26	2.73	3.57	6.57	4.49	26.11	2.86	3.34	41.43
3.5	1.25	3.21	7.52	5.52	1.00	2.80	5.35	16.50	6.11
4.5	3.42	1.77	4.80	8.81	16.69	13.48	2.62	22.20	27.90
5.5	3.56	6.50	2.65	9.02	15.37	41.44	16.87	34.64	77.79
6.5	3.02	1.75	16.28	2.37	17.28	10.32	41.97	41.46	4.13
7.5	1.04	7.08	40.05	18.08	8.41	4.51	20.31	26.49	13.39
8.5	1.54	45.49	32.24	12.80	1.87	164.04	12.87	5.88	18.70
9.5	3.50	14.20	5.95	22.68	31.77	16.06	23.15	11.59	9.14
$\gamma = 0.7$									
0.5	1.00	1.50	1.40	4.96	1.56	2.43	2.69	4.25	7.28
1.5	2.01	2.06	5.62	3.35	2.45	5.79	1.41	12.21	5.26
2.5	1.97	5.19	5.19	3.20	4.87	28.52	6.81	2.85	16.75
3.5	3.54	17.68	7.98	9.97	8.43	6.20	8.01	43.74	40.68
4.5	3.52	1.03	20.75	4.30	17.94	6.41	19.85	2.23	39.76
5.5	1.24	4.09	8.23	11.29	17.45	16.77	36.87	23.95	13.17
6.5	2.99	7.08	5.39	14.58	7.54	6.34	11.00	24.89	50.57
7.5	8.17	6.96	11.81	10.06	39.66	24.75	75.61	16.55	20.20
8.5	7.08	7.88	14.15	42.45	66.46	37.90	23.04	13.01	63.03
9.5	13.32	14.31	21.61	41.20	9.86	50.32	8.93	85.37	64.13

Table 3: Experiment I: results.

tend to increase as parameter λ increases. For example, in the third part of Table 3, $r(0.5, 10) = 1.00$, while $r(9.5, 10) = 13.32$. The response time deviations increase with γ, as well. On average, the $r(\lambda, N)$ values in Table 3 that correspond to $\gamma = 0.1$ are lower than the values for $\gamma = 0.4$, which, in turn, are lower than the values for $\gamma = 0.7$ in Table 3.



(a) (b)

Figure 3: Results for (a)Experiment II and (b)Experiment III . The dashed horizontal line corresponds to the case where OLPC and Greedy exhibit similar performance.

The aforementioned observations are further confirmed by the following experiment, the results of which are shown in Fig. 3(a). In the figure, the x-axis corresponds to the λ values, $\lambda \in \{0.25, 0.5, \ldots, 10\}$, while the y-axis corresponds to the aggregate response time ratio values $\overline{r}$. For every $\lambda \in \{0.25, 0.5, \ldots, 10\}$ and $N \in \{10, 20, 30, \ldots, 250\}$ value, a simulation instance is set up. In this instance, $\gamma = 0.1$ and $\boldsymbol{\sigma} \sim \mathbf{U}_{\sigma}(0, 1)$, i.e. the resulting instances are of the form $[\mathbf{G_c}(10, 5), \mathbf{G_t}(10\lambda, 10\lambda 0.1), \mathbf{U}_{\sigma}(0, 1), 0, N]$. The aggregate response time ratio $\overline{r}_{\gamma=0.1}(\lambda)$ values are given by:

$$\overline{r}_{\gamma=0.1}(\lambda) = \frac{\sum_{\forall N} \rho'_{[\mathbf{G_c}(10,5),\mathbf{G_t}(10\lambda,10\lambda 0.1),\mathbf{U}_{\sigma}(0,1),0,N]}}{\sum_{\forall N} \rho_{[\mathbf{G_c}(10,5),\mathbf{G_t}(10\lambda,10\lambda 0.1),\mathbf{U}_{\sigma}(0,1),0,N]}}$$

The same procedure is repeated for $\gamma = 0.4$ and $\gamma = 0.7$. In essence, $\overline{r}(\lambda) = 1$ means that all plans built by Greedy have exactly the same response times as the plans built by OLPC. From Fig. 3(a) we observe that the aggregate response time ratio values, and consequently, the response time deviations between the Greedy and OLPC plans, increase as the value of parameter λ increases, i.e., as the communication costs become more dominant. For example, consider the $\overline{r}_{\gamma=0.1}(\lambda)$ values. Fig. 3(a) shows that the plans built by Greedy have up to 4.9 times higher response time than the ones built by OLPC for $\lambda \in \{0.25, 0.5, \ldots, 5\}$. On the other hand, for $\lambda \in \{5, 5.25, \ldots, 10\}$, this deviation reaches 7.8 times. Recall that an increase in λ entails a deviation of $\overline{t}$ from $\overline{c}$. Analogous observations can be drawn for $\overline{r}_{\gamma=0.4}(\lambda)$ and $\overline{r}_{\gamma=0.7}(\lambda)$, too.

In summary, from Fig. 3(a) and Table 3, we can see that, for fixed γ, the plans built by Greedy have response time very close to the response time of the plans built by OLPC when the mean transferring cost per tuple $\bar{t}$ is lower than (or close to) the mean processing cost per tuple $\bar{c}$, independently of the number of input services. Furthermore, we can observe that as the standard deviation σ_t of $\mathbf{t}$ increases, i.e. the network heterogeneity increases, the response time deviations between the Greedy and the OLPC plans increase, as well. For example, when $\sigma_t = 0.4\bar{t}$, the response times of Greedy plans are at most 16.7 times higher than the response time of OLPC plans. On the other hand, when $\sigma_t = 0.7\bar{t}$ the maximum deviation is up to 26.1 times, while for $\bar{t} = 0.1\bar{c}$, the maximum deviation does not exceed the 7.8 times. The explanation is that when the network heterogeneity is limited, the costs needed to transfer tuples between any pair of WSs are approximately the same. As mentioned in a previous section, the Greedy algorithm can optimally solve a special case of Problem 1.1, when one WS sends tuples to the rest WSs with the same speed. However, as heterogeneity increases, the performance of the Greedy algorithm degrades significantly.

Experiment III: impact of selectivities on performance. In the previous experiments, the WSs selectivities σ_i were uniformly distributed in the interval $(0\ 1)$. To investigate the impact of the selectivity values in more detail, we conduct the following experiment. For each λ and N value a simulation instance (γ is set to 0.7) is set up. We test four cases for σ: $\mathbf{U}_\sigma(0,1)$, $\mathbf{U}_\sigma(0,0.5)$, $\mathbf{U}_\sigma(0.5,0.8)$ and $\mathbf{U}_\sigma(0.8,1)$. The corresponding aggregate response time ratio values $\overline{r}(\lambda)$ are shown in Fig. 3(b). When $\boldsymbol{\sigma} \sim \mathbf{U}_\sigma(0.8,1)$, the maximum deviation reaches 14.4 times, while when $\boldsymbol{\sigma} \sim \mathbf{U}_\sigma(0,1)$ it reaches 26.1 times. In general, the results of this experiment show that when $\boldsymbol{\sigma} \sim \mathbf{U}_\sigma(0,0.5)$, $\boldsymbol{\sigma} \sim \mathbf{U}_\sigma(0.5,0.8)$ or $\boldsymbol{\sigma} \sim \mathbf{U}_\sigma(0.8,1)$ the response time deviations are similar. However, when $\boldsymbol{\sigma} \sim \mathbf{U}_\sigma(0,1)$ the deviations become much higher. Thus, for fixed λ, γ and N values, the deviation of the response time of the Greedy algorithm from the optimal increases as the range of the services' selectivities increases.

Experiment IV: comparison against variants of the Greedy algorithm. Since the Greedy algorithm considers only the processing costs of tuples in order to build a plan,

$\lambda \setminus N$	10	40	70	100	130	160	190	220	250
Min-Greedy									
0.5	1.00	1.50	1.40	3.41	1.56	2.43	2.69	4.25	7.28
1.5	2.01	4.86	3.05	18.81	8.61	11.40	17.56	12.21	4.57
2.5	1.00	6.26	7.07	5.22	2.49	13.93	8.64	33.08	44.65
3.5	1.60	4.93	15.84	7.17	13.69	23.76	19.47	27.37	3.75
4.5	3.47	3.91	3.71	16.57	9.47	4.57	14.89	46.84	30.81
5.5	7.03	2.38	8.23	9.69	21.51	30.95	19.43	103.76	42.75
6.5	7.59	15.48	56.04	30.22	6.47	76.85	20.75	21.48	50.57
7.5	5.12	8.22	29.89	14.29	39.66	6.00	1.91	25.24	14.63
8.5	2.57	14.83	13.48	37.79	28.22	5.26	8.81	13.70	63.03
9.5	4.39	12.96	15.88	55.24	49.81	5.76	27.40	14.23	58.48
Max-Greedy									
0.5	1.83	3.73	2.24	3.98	1.35	2.07	3.28	5.52	6.78
1.5	1.00	1.53	2.35	16.52	8.47	3.96	42.45	10.72	2.84
2.5	7.54	4.43	4.98	4.39	7.39	19.15	6.06	13.25	19.32
3.5	3.01	4.70	9.86	45.94	13.81	30.67	30.43	9.63	22.28
4.5	5.39	19.59	6.59	13.68	32.32	10.02	13.61	129.64	7.12
5.5	2.44	9.49	21.09	18.40	7.19	13.91	7.01	42.79	34.81
6.5	3.79	8.61	31.23	47.17	109.76	22.47	6.31	7.86	89.57
7.5	5.12	7.31	28.75	17.56	17.02	12.88	10.16	39.32	175.40
8.5	8.05	11.86	16.17	38.99	102.79	56.71	50.19	45.93	106.83
9.5	3.82	10.51	9.57	58.59	8.87	37.89	42.72	41.93	44.21
Mean-Greedy									
0.5	1.00	1.50	1.40	2.92	1.56	2.43	2.69	4.25	5.46
1.5	1.69	2.06	8.01	22.37	2.45	5.47	17.56	5.54	24.78
2.5	1.20	4.59	5.19	2.80	9.49	6.03	9.02	3.71	39.62
3.5	3.09	4.37	2.33	5.20	6.61	17.60	4.36	18.97	64.05
4.5	5.39	11.93	8.02	2.72	18.67	20.16	25.13	2.15	12.07
5.5	1.46	16.56	2.90	11.29	9.63	2.86	27.77	60.51	23.46
6.5	2.75	6.04	14.37	39.74	7.23	104.20	3.30	3.21	3.30
7.5	5.12	6.15	5.73	9.61	16.64	22.52	1.37	22.12	30.54
8.5	5.16	6.33	10.64	48.34	3.77	14.32	25.76	82.70	5.99
9.5	5.08	11.97	10.34	42.75	18.33	22.94	11.86	82.50	84.97

Table 4: Experiment IVa: experiment I ($\gamma = 0.7$) is redone for all variants.

Var. $\setminus \lambda$	0.5	1.5	2.5	3.5	4.5	5.5	6.5	7.5	8.5	9.5
Greedy	1.80	4.82	6.58	6.62	8.93	9.54	9.89	11.68	11.40	15.24
Min-	1.89	4.71	6.23	7.75	7.71	7.02	11.96	12.29	15.54	17.34
Max-	2.31	5.42	6.38	8.64	12.93	9.06	11.54	15.59	16.04	18.52
Mean-	1.88	4.44	6.15	5.65	7.97	7.78	9.08	11.75	13.24	14.25

Table 5: Experiment IVb: experiment II ($\gamma = 0.4$) is redone for all variants.

Var. \ λ	0.5	1.5	2.5	3.5	4.5	5.5	6.5	7.5	8.5	9.5
$\boldsymbol{\sigma} \sim \mathbf{U}_\sigma(0.8\ 1)$										
Greedy	2.38	4.22	5.75	7.35	7.61	8.40	9.98	11.20	12.68	13.18
Min-	2.45	4.27	6.51	7.05	7.55	9.06	9.70	10.55	12.47	13.69
Max-	2.28	4.43	5.60	7.17	7.69	8.91	10.34	11.70	12.50	12.91
Min-	2.37	4.12	5.33	7.01	7.09	8.34	8.52	10.93	10.28	11.86
$\boldsymbol{\sigma} \sim \mathbf{U}_\sigma(0\ 1)$										
Greedy	2.44	4.84	7.43	10.21	10.37	13.89	14.24	17.33	23.09	19.37
Min-	2.40	5.78	8.44	10.15	10.75	14.55	16.40	13.59	18.91	20.08
Max-	2.64	5.95	11.03	9.89	14.60	16.30	16.97	16.26	25.90	18.77
Min-	2.54	5.02	7.70	7.61	12.08	11.67	12.72	12.39	19.49	16.25

Table 6: Experiment IVc: experiment III ($\gamma = 0.7$) is redone for all variants.

we have implemented three simple variants that do take into consideration the transferring costs of tuples. A simple way to expand Greedy is to tweak the processing costs of tuples for each input WS. Let v_i be the new processing cost per tuple of the service WS_i. In the first variant of Greedy, the Min-Greedy, the per tuple processing cost v_i of the service WS_i is set to $v_i = c_i + \min_j\{t_{i,j}\}$. Similarly, in Max-Greedy and Mean-Greedy variants, the per tuple processing costs of WS_i is set to $v_i = c_i + \max_j\{t_{i,j}\}$ and $v_i = c_i + \sum_{j \neq i} t_{i,j}/(N-1)$, respectively. We have repeated experiments I, II, and III and the results are shown in Tables 4, 5 and 6, respectively. We will not move to a detailed analysis of the experimental results due to lack of space. However, it must be noted that none of the three variants can significantly improve the performance of Greedy and that Max-Greedy may lead to more severe performance degradations.

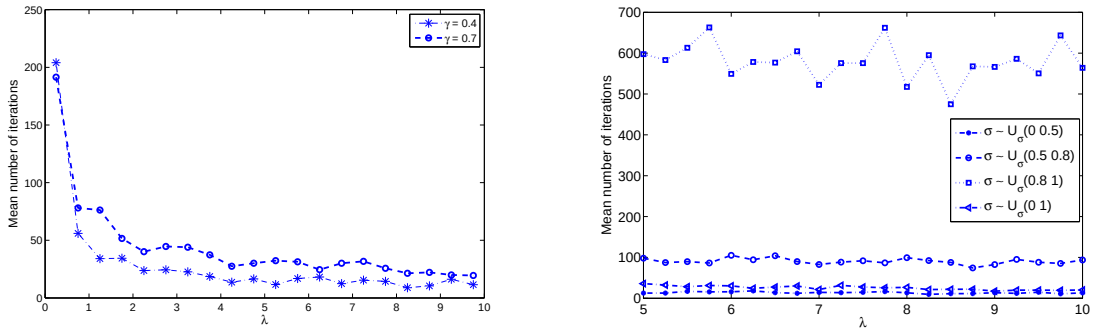
Experiment V: absolute running time and number of iterations. For all the experiments, we used a machine with a dual core processor. The CPU clock of each core is at 2.00 GHz and the total memory 2GB. The mean running time per simulation instance is only 0.3 sec, which can be deemed as negligible[3] and constitutes a strong proof of the efficiency of the algorithm.

The claim about the efficiency of the algorithm is further supported by the following experiments that aim at studying the number of iterations under different conditions. Following the same approach as in Experiments I-III, we vary the number of services,

[3]WSs typically spend time in the order of seconds when processing chunks of data; see, for example, the real measurements mentioned in [10].

$\lambda \backslash N$	10	40	70	100	130	160	190	220	250
$\gamma = 0.4$									
0.5	3	65	91	61	11	114	74	12	335
1.5	4	3	18	45	24	139	86	49	28
2.5	3	7	7	35	22	45	22	20	16
3.5	4	14	12	5	9	13	37	25	22
4.5	3	3	23	19	16	29	19	31	66
5.5	18	3	5	10	6	4	12	33	5
6.5	3	3	3	5	14	29	6	4	8
7.5	3	16	4	6	15	4	20	32	43
8.5	4	3	4	4	4	4	15	8	16
9.5	4	12	5	10	10	5	31	4	16
$\gamma = 0.7$									
0.5	6	72	58	111	12	117	325	105	201
1.5	8	4	21	6	8	73	3	110	52
2.5	4	16	37	26	8	47	26	9	58
3.5	7	5	18	15	18	53	43	35	6
4.5	10	5	17	68	64	53	146	7	32
5.5	5	20	10	49	46	24	6	25	54
6.5	7	45	9	17	20	11	49	19	73
7.5	4	6	4	51	17	14	6	49	40
8.5	13	4	28	22	25	19	21	21	25
9.5	3	13	6	4	33	4	30	17	47

Table 7: Experiment V: number of iterations for different values of N and λ.



(a) (b)

Figure 4: Results for Experiment V. (a)Average number of iterations for different values of λ and (b)average number of iterations for different distributions of σ.

the ratio of processing and communication cost and the selectivity values. First, we conduct an experiment in an environment similar to that of Experiment I. As such, for each λ and N, the OLPC algorithm is executed in simulation instances of the form

$\lambda \backslash N$	10	40	70	100	130	160	190	220	250
$\gamma = 0.4$, $\boldsymbol{\sigma} \sim \mathbf{U}_{\sigma}(0.8\ 1)$									
0.5	903	675	1176	979	2221	2414	1420	3450	3246
1.5	80	346	265	564	553	532	1119	829	899
2.5	38	437	278	624	443	875	1016	692	1089
3.5	111	256	137	264	441	420	455	628	798
4.5	187	244	354	175	500	649	543	645	1074
5.5	258	166	247	130	250	454	800	673	447
6.5	51	246	393	346	537	354	545	469	797
7.5	740	215	298	446	412	379	381	523	823
8.5	97	235	189	449	447	307	596	714	707
9.5	93	232	231	244	399	487	424	409	341
$\gamma = 0.7$, $\boldsymbol{\sigma} \sim \mathbf{U}_{\sigma}(0.8\ 1)$									
0.5	164	1424	529	2687	5089	1605	2049	4143	3738
1.5	66	403	400	579	897	1040	923	2246	2088
2.5	162	191	510	716	510	1328	1316	916	1995
3.5	536	325	179	526	485	1062	1200	769	1124
4.5	59	308	313	587	458	615	903	986	981
5.5	65	254	553	322	756	687	1181	607	932
6.5	1149	383	196	482	374	852	524	500	1094
7.5	175	371	325	371	684	626	933	501	832
8.5	107	400	183	395	424	459	558	624	925
9.5	200	191	271	795	691	608	841	849	1441

Table 8: Experiment V: number of iterations for different values of N and λ when selectivity is high.

$[\mathbf{G_c}(10, 5), \mathbf{G_t}(10\lambda, 10\lambda\gamma),\ \mathbf{U}_{\sigma}(0, 1), 0, N]$. After every simulation, the number of iterations $\zeta(\lambda, N)$ is counted. The $\zeta(\lambda, N)$ values for $\lambda \in \{0.5, 1.5, \ldots, 9.5\}$, $N \in \{10, 40, \ldots, 250\}$, and $\gamma \in \{0.4,\ 0.7\}$ are shown in Table 7. In general, we observe that for fixed λ and γ, the number of iterations that the OLPC algorithm performs is correlated with the number of the input services, but, on average, it increases slowly. Also, in cases where it happens the costs of some services to lie at the left tail of the distribution and their selectivities to be low, the number of iterations is very small, since it suffices to examine a few of orderings of only these services. This explains the very low values appearing in Table 7.

The network heterogeneity which relates to the values of γ has an impact, too. For example, on average, $\zeta(\lambda, N)$ values of the upper part of the Table 7 are lower than the

values of lower part of the table. Similar observations can be drawn from Fig. 4(a). This figure shows the mean number of iterations $\overline{\zeta}(\lambda)$ for $\lambda \in \{0.25, 0.5, \ldots, 10\}$ for all values of $N \in \{10, 20, 30, \ldots, 250\}$. The $\overline{\zeta}(\lambda)$ values are given by:

$$\overline{\zeta}(\lambda) = \frac{\sum_{\forall N} \zeta_{[\mathbf{G_c}(10,5),\mathbf{G_t}(10\lambda,10\lambda\gamma),\mathbf{U}_\sigma(0,1),0,N]}}{|N|}$$

From Fig. 4(a), we see that the mean number of iterations decreases as the ratio $\overline{t}/\overline{c}$ increases. For example, the mean number of iterations is 192 when $\lambda = 0.25$ and $\gamma = 0.7$, while this value rapidly decreases as λ approaches 10. Also, from the figure it is clear that the number of iterations increases as the network heterogeneity (i.e., parameter γ) increases.

Finally, we check the impact of the distribution of the WS selectivities σ_i on the number of iterations performed by OLPC. To this end, an experiment similar to Experiment III is executed; however, the output of interest is the mean number of iterations $\overline{\zeta}(\lambda)$. The experimental results are shown in Fig. 4(b). We see that the number of iterations seems to depend, to a large extent, on the selectivities' distribution. When $\boldsymbol{\sigma} \sim \mathbf{U}_\sigma(0.8, 1)$ the number of iterations is higher than any other selectivities distribution for a given simulation environment. For the rest three $\boldsymbol{\sigma}$ distributions the number of iterations is approximately the same. This phenomenon can be explained by the following fact: when $\boldsymbol{\sigma} \sim \mathbf{U}_\sigma(0.8, 1)$ the value of variable $\overline{\epsilon}$ decreases at a slower rate, so that the expansion phase needs more iterations. As a result, the algorithm converges to the optimal in more iterations. Table 8 shows the number of iterations for examples with high selectivities in more detail. For fixed values of λ, the average increase in the number of iteration with regards to the number of services is still linear.

4.3 Precedence constraints

The last part of the evaluation consists of experiments with WSs for the case when there are precedence constraints among them. In [24], it is described how WS plans respecting any such constraints are built in $O(n^5)$ time. We re-execute the experiments of the previous subsection after having modified the values of parameter p. We experiment with two

$\lambda \setminus N$	10	40	70	100	130	160	190	220	250
$p = 0.4$									
0.5	1.02	1.27	1.11	1.03	1.24	1.13	1.07	1.75	1.06
1.5	1.61	1.70	1.38	1.47	2.01	1.57	1.59	1.21	1.49
2.5	2.31	1.38	1.06	4.09	6.19	3.86	2.17	1.01	1.84
3.5	1.37	1.14	3.38	8.31	2.62	3.09	1.21	3.44	4.21
4.5	1.57	1.62	3.41	3.79	6.79	24.38	1.97	2.40	4.17
5.5	12.68	4.47	2.34	6.40	2.20	6.78	1.95	6.10	6.75
6.5	1.44	1.44	4.76	4.14	1.44	3.61	1.21	1.20	4.56
7.5	1.00	4.76	3.23	5.23	2.08	1.40	8.87	2.32	1.57
8.5	3.12	2.65	2.55	10.63	2.77	1.85	1.06	14.73	4.92
9.5	1.61	1.25	9.52	5.89	2.37	2.37	8.01	1.43	5.92
$p = 0.6$									
0.5	1.28	1.00	1.16	1.00	1.01	1.01	1.00	1.00	1.00
1.5	2.39	1.00	1.19	1.29	1.00	3.41	1.42	1.03	1.00
2.5	2.65	1.00	1.00	1.15	1.06	1.00	1.00	1.15	1.50
3.5	1.32	1.89	1.02	1.00	1.01	2.55	1.12	1.13	1.18
4.5	1.00	1.00	1.12	1.00	1.00	1.29	1.00	1.15	1.00
5.5	1.80	1.40	1.00	1.00	1.00	1.41	1.00	1.13	1.08
6.5	1.00	1.00	1.65	1.12	1.00	2.19	1.45	1.00	1.89
7.5	4.48	1.92	1.00	1.27	1.00	2.41	1.34	1.00	2.79
8.5	1.00	1.83	1.21	2.41	1.76	2.66	1.00	1.04	1.00
9.5	1.00	1.37	1.12	1.00	1.58	1.00	1.04	1.78	1.13

Table 9: Experiment VI: results for the same environment as Experiment I.

values of p, $p = 0.4$ and $p = 0.6$, both of which result in a high number of precedence constraints; results for lower values of p, e.g., 0.1, are omitted because they are very similar to the results for the case without precedence constraints. In all constrained plans, a single WS plays the role of data generator, i.e., that WS must precede all other WSs. As explained earlier, that parameter p controls the probability according to which a random service WS_i is prerequisite for another service $WS_j, i < j$.

Experiment VI: impact of p on performance. Table 9 shows the experimental results derived from Experiment I when $p = 0.4$ and $p = 0.6$ ($\gamma = 0.7$). Although the response time deviations increase as the number of input WSs increases, as the value of parameter p increases, they tend to diminish. This is actually expected and is explained by the fact that the number of possible plans that can be constructed shrinks with high p

γ / λ	0.5	1.5	2.5	3.5	4.5	5.5	6.5	7.5	8.5	9.5
$p = 0.4$										
0.4	1.13	1.58	1.85	2.15	2.14	2.34	3.18	2.67	2.91	3.42
0.7	1.28	1.65	2.22	2.63	3.57	4.45	3.26	3.92	3.76	3.81
$p = 0.6$										
0.4	1.09	1.19	1.14	1.14	1.24	1.29	1.25	1.22	1.53	1.33
0.7	1.08	1.11	1.26	1.39	1.21	1.26	1.44	1.67	1.53	1.20

Table 10: Experiment VI: part of the results for the same environment as Experiment II.

probabilities. In fact, in highly constrained environments, the plan construction process is essentially determined by the constraints. This becomes more clear when $p = 0.6$, where the Greedy algorithm tends to produce equivalent plans with OLPC, despite the network disparity and the processing and the transferring costs of the simulation instances. For $p = 0.4$ and $N = 250$, the maximum value of r is 6.75, while for $p = 0.6$ and $N = 250$, the maximum value of r decreases to 2.79. In other words, OLPC can still improve the performance by several factors. However, the maximum value of r for $\gamma = 0.7$ and $N = 250$ when $p = 0$ (unconstrained case) is 64.1, as shown in Table 3.

We also regenerate the environment of Experiment II. The aggregate response time ratio values are similar to the results shown in Figure 3(a), i.e. the response time deviations between the $Greedy$ and the OLPC plans increase, as the network heterogeneity increases (the figure is omitted). A part of the results is presented in Table 10. For $p = 0.4$ and $\gamma = 0.4$, $\overline{r}(8.5) = 2.91$, while for $p = 0.4$ and $\gamma = 0.7$, $\overline{r}(8.5)$ increases to 3.76. The corresponding values for $p = 0$ (unconstrained case) are 11.2 and 23.9, for $\gamma = 0.4$ and $\gamma = 0.7$, respectively (see Fig. 3(a)).

Experiment VII: impact of p on efficiency. We conclude are experiments with the investigation of the impact of p on the number of iterations. As expected, the iterations the OLPC algorithms performs in order to complete decreases as parameter p increases. The reason behind that fact is quite clear. As stated in the previous paragraph, the number of possible plans is much lower in a constrained environment. Related to that, the termination condition of the OLPC algorithm checks fewer plans, as some WS couples are not

$\gamma \setminus \lambda$	1.5	2.5	3.5	4.5	5.5	6.5	7.5	8.5	9.5
$p = 0.4$									
0.4	3.16	5.12	8.52	21.44	3.76	3.88	24.64	3.36	19.6
0.7	14.93	13.44	5.57	27.13	24.16	23.96	4.48	4.56	5.12
$p = 0.6$									
0.4	5.64	6.68	6.83	6.40	5.84	6.64	7.52	6.72	7.21
0.7	6.72	7.12	7.38	6.76	7.36	7.72	6.72	6.96	6.83

Table 11: Experiment VII: number of iteration for different values of λ and γ when $p = 0.4$ and $p = 0.6$.

$\lambda \setminus N$	10	40	70	100	130	160	190	220	250
$p = 0.4$									
0.5	12	13	16	30	13	8	15	76	10
1.5	15	99	11	18	19	19	229	26	32
2.5	8	28	58	55	14	494	15	132	18
3.5	75	66	308	86	56	20	22	130	28
4.5	11	33	22	21	46	53	54	47	45
5.5	10	17	194	19	25	25	52	75	29
6.5	860	21	24	17	462	32	51	113	28
7.5	24	24	246	363	106	79	24	29	35
8.5	10	21	168	46	31	102	33	32	64
9.5	64	45	163	60	99	32	25	132	25
$p = 0.6$									
0.5	11	5	9	7	15	13	22	9	21
1.5	25	17	30	13	14	18	18	15	31
2.5	19	35	34	10	24	19	13	27	15
3.5	10	132	55	17	21	10	31	16	18
4.5	10	73	45	21	26	13	29	61	24
5.5	8	22	18	52	14	20	14	10	20
6.5	12	22	12	18	33	18	48	31	30
7.5	9	14	20	21	11	20	30	104	11
8.5	12	17	9	42	31	150	16	19	89
9.5	10	11	99	16	29	12	25	40	39

Table 12: Experiment VII: number of iteration for different values of λ and N when $p = 0.4$ or $p = 0.6$, $\gamma = 0.7$ and $\sigma \sim \mathbf{U}_\sigma(0.8, 1)$.

valid plan prefixes. Recall that in these experiments, a source WS always exist, which corresponds to the source of input data, so that it is must be first service in every plan $\mathcal{C}$. Thus, the number of WS couples (WS_l, WS_r) that can be placed in the first two positions of a serial plan $\mathcal{C}$, i.e. $WS_l = WS_{k(0)}$ and $WS_r = WS_{k(1)}$, is $N - 1$. In contrast, in an unconstrained environment the number of such possible couples is $N(N - 1)$, i.e., the

plan search space is significantly narrower.

Table 11 presents the experimental results after we rerun Experiment V for $p = 0.4$ and $p = 0.6$. It is clear, that the latter have many similarities with the results of Fig. 4(a), i.e. the number of performed iterations increases when the network disparity increases; however, it decreases as parameter p increases. For example, for $p = 0.4$ and $\gamma = 0.4$, $\overline{\zeta}(9.5) = 19.6$, while for $p = 0.6$ and $\gamma = 0.4$, $\overline{\zeta}(9.5)$ decreases to 7.21. However, the number of iterations is generally higher for $p = 0$ (see Fig. 4(a)). We also experiment with the impact of the selectivity values. The corresponding number of iterations $\zeta(\lambda, N)$ are shown in Table 12. The number of iterations increases when the number of input WSs increases, too. However, as explained above, this number is much smaller, comparably with the results of the second part of Table 8.

5 Related Work

Our work relates to the broader areas of distributed query optimization and pipelined operator ordering. Distributed query optimization algorithms differ from their centralized counterparts in that communication cost must be considered and there is a trade-off between total work optimization and the harder problem of response time optimization [9]. Proposals for the latter case either employ more sophisticated dynamic programming techniques (e.g., [14]) or resort to heuristics. Response time optimization is largely affected by the types of parallelism in the query plan; nevertheless, typically only independent parallelism is investigated in wide area settings (e.g., [13, 7]), whereas our focus is on pipelined parallelism. There is a lot of work for different settings than the one assumed in this work (e.g., P2P networks [21, 12]); however such proposals do not share the same goals and cannot be applied to our problem.

Pipelined operator ordering has been examined for both centralized and distributed environments. In a centralized single-node environment, the problem of minimizing the response time can be optimally solved in polynomial time only if the selectivities are independent [15, 11]; note that if the independence assumption does not hold the prob-

lem becomes intractable. In a wide-area environment, the response time optimization problem is transformed to bottleneck cost minimization. In this setting, Srivastava *et al* [24] proposed an algorithm for optimizing select-project-join queries over WSs. However, they assume that the query execution process is simplified through a WSMS which orchestrates data exchange among the services, so that joins can be computed and the heterogeneity of communication costs does not impact on the bottleneck cost metric. As such, our algorithm can be deemed as an extension to [24] for the case when decentralized sequential plans are examined; note that [24] support parallel plans as well, which outperform sequential ones when service selectivities are higher than 1. Braga *et al* [4] deal with a slightly different problem, where IR-style tasks are combined with accurate search tasks in the same query; the goal is to produce a WS invocation plan (either sequential or parallel) in order to obtain the best k answers of a query in the presence of access limitations but the algorithm they employ involves exhaustive search of the candidate plans. Deshpande *et al* [6] consider correlated selective attributes but they aim at minimizing the total cost for acquiring the values of the attributes, since they assume that each attribute is assigned an acquisition cost. The plan constructed is a conditional one in the form of a binary decision tree. All these proposals are static. In [3], the goal is to develop solutions for the ordering of selective operators that are tailored to online, dynamic scenarios. However, the approximate algorithm in [3] applies only to the problem of minimizing the total work and assumes selectivities not higher than 1. Complementarily to the above, [23] explore adaptive approaches to parallelizing calls to WSs.

None of these works consider the communication costs. Existing solutions for multi-query optimization neglect the communication costs, as well. E.g., [20, 18] assume a single-node execution environment, where all operators are selective, potentially correlated and unconstrained. The optimization metric is the minimization of the sum of the operator costs, as in a distributed version of the same problem discussed in [17].

A common characteristics of the proposals mentioned so far is that they build a single plan. For completeness, we mention techniques that define a set of interleaving plans in order to maximize the data flow, which is equivalent to minimizing the bottleneck cost. In

[5], such a tuple routing algorithm is proposed in order to maximize the flow of tuples processed by the filters of the input query. The filters are all selective and unconstrained. The output is a set of serial plans. Each serial plan is assigned a probability weight and when a new tuple enters the system, it is assigned to one of these serial plans with a probability depending on its weight. It must be noted that the flow maximization algorithm considers only the process rates of the filters (i.e., the number of tuples per unit of time) and the potentially heterogeneous communication costs are disregarded. This work is extended in [8] to also support proliferative operators and precedence constraints. However, [8] is characterized by the limitation of not considering communication costs, too. Note that interleaving plans are not the same as eddies [2, 25]; the former deal with multiple static plans, whereas the latter refer to a single plan that is continuously adapted to changes in the environment.

Finally, a recent work that takes data transmission into account has appeared in [16], which deals with processing of multiple, overlapping, non-parallel queries. The input data is in sources stored on different host machines, while the cost to transfer data between any two hosts varies, as in our problem. Nevertheless, the optimization goal is different; the algorithm in [16] aims to minimize the total cost to transfer data across overlapping queries, whereas we focus on minimizing the response time of a pipelined parallel query.

6 Conclusions and Directions for Future Work

In this work, we deal with the optimization of decentralized queries over Web Services. More specifically, we present an algorithm for finding the optimal ordering of pipelined services when the services communicate directly with each other and the communication costs vary. The goal is to minimize query response time, which, due to parallelism, depends on the bottleneck service in the plan. Our algorithm operates regardless of any precedence constraints and selectivity values can be higher than 1. To the best of our knowledge, it is the first attempt to solve this intractable problem. Our algorithm is provably optimal, i.e., always finds the optimal plan, and particularly efficient in terms of

running time, as the results of the thorough evaluation reveal. It follows the branch and bound optimization approach and adopts a novel pruning technique in order to reduce the search space. It can yield performance improvements of an order of magnitude in realistic scenarios.

Our work has been motivated by emerging paradigms of distributed data management and can be extended towards several directions in order to fully fulfill modern needs. In the future, we plan to investigate solutions that support more generic plans rather than more simple sequential orderings of operators. In such plans, each service can have multiple inputs and disseminate its results to multiple services simultaneously. The investigation of correlated selectivities and the development of adaptive flavors of the algorithm is also left for future work. Finally, we believe that, in distributed settings, operator ordering solutions must be coupled with resource allocation and scheduling algorithms in order to produce a complete solution. We plan to work to this end in the future, too.

References

[1] M. N. Alpdemir, A. Mukherjee, A. Gounaris, N. W. Paton, P. Watson, A. A. A. Fernandes, and D. J. Fitzgerald. Ogsa-dqp: A service for distributed querying on the grid. In *EDBT*, 2004.

[2] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. In *SIGMOD*, pages 261 – 272, 2000.

[3] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD*, pages 407–418, 2004.

[4] D. Braga, S. Ceri, F. Daniel, and D. Martinenghi. Optimization of multidomain queries on the web. *VLDB Endowment*, volume 1, pages 562 – 573, 2008.

[5] A. Condon, A. Deshpande, L. Hellerstein, and N. Wu. Algorithms for distributional and adversarial pipelined filter ordering problems. *ACM Transactions on algorithms*, 5(2):24 – 34, 2009.

[6] A. Deshpande, C. Guestrin, W. Hong, and S. Madden. Exploiting correlated attributes in acquisitional query processing. In *ICDE*, pages 143 – 154, 2005.

[7] A. Deshpande and J. M. Hellerstein. Decoupled query optimization for federated database systems. In *ICDE*, pages 716–732, 2002.

[8] A. Deshpande and L. Hellerstein. Flow algorithms for parallel query optimization. In *ICDE*, pages 754 – 763, 2008.

[9] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *SIGMOD*, pages 9–18, 1992.

[10] A. Gounaris, C. Yfoulis, R. Sakellariou, and M. D. Dikaiakos. Robust runtime optimization of data transfer in queries over web services. In *ICDE*, pages 596–605, 2008.

[11] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *SIGMOD*, pages 267–276, 1993.

[12] R. Huebsch, B. N. Chun, J. M. Hellerstein, B. T. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A. R. Yumerefendi. The architecture of pier: an internet-scale query processor. In *CIDR*, pages 28–43, 2005.

[13] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[14] D. Kossmann and K. Stocker. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Transactions on Database Systems*, 25(1):43–82, 2000.

[15] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *VLDB*, pages 128–137, 1986.

[16] J. Li, A. Deshpande, and S. Khuller. Minimizing communication cost in distributed multi-query processing. In *ICDE*, pages 772–783, 2009.

[17] Z. Liu, S. Parthasarathy, A. Ranganathan, and H. Yang. Generic flow algorithm for shared filter ordering problems. In *PODS*, pages 79 – 88, 2008.

[18] Z. Liu, S. Parthasarathy, A. Ranganathan, and H. Yang. Near-optimal algorithms for shared filter evaluation in data stream systems. In *SIGMOD*, pages 133 – 146, 2008.

[19] T. Malik, A. S. Szalay, T. Budavari, and A. Thakar. Skyquery: A web service approach to federate databases. In *CIDR*, 2003.

[20] K. Munagala, U. Srivastava, and J. Widom. Optimization of continuous queries with shared expensive filters. In *PODS*, pages 215 – 224, 2007.

[21] W. S. Ng, B. C. Ooi, K.-L. Tan, and A. Zhou. Peerdb: A p2p-based system for distributed data sharing. In *ICDE*, pages 633–644, 2003.

[22] M. A. Nieto-Santisteban, J. Gray, A. S. Szalay, J. Annis, A. R. Thakar, and W. O'Mullane. When database systems meet the grid. In *CIDR*, pages 154–161, 2005.

[23] M. Sabesan and T. Risch. Adaptive parallelization of queries over dependent web service calls. In *ICDE*, pages 1725–1732, 2009.

[24] U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query optimization over web services. In *VLDB*, pages 355 – 366, September 2006.

[25] F. Tian and D. J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, pages 333–344, 2003.