Distributed and Parallel Databases (2019) 37:385–410
https://doi.org/10.1007/s10619-018-7243-3

Optimization of data flow execution in a parallel environment

Georgia Kougka[1] · **Anastasios Gounaris**[1]

Published online: 22 August 2018

Abstract

Although the modern data flows are executed in parallel and distributed environments, e.g. on a multi-core machine or on the cloud, current cost models, e.g., those considered by state-of-the-art data flow optimization techniques, do not accurately reflect the *response time* of real data flow execution in these execution environments. This is mainly due to the fact that the impact of parallelism, and more specifically, the impact of concurrent task execution on the running time is not adequately modeled in current cost models. The contribution of this work is twofold. Firstly, we propose an advanced cost model that aims to reflect the *response time* of a data flow that is executed in parallel more accurately. Secondly, we show that existing optimization solutions are inadequate and develop new optimization techniques targeting the proposed cost model. We focus on the single multi-core machine environment provided by modern business intelligence tools, such as Pentaho Kettle, but our approach can be extended to massively parallel and distributed settings. The distinctive features of our proposal is that we model both time overlaps and the impact of concurrency on task running times in a combined manner; the latter is appropriately quantified and its significance is exemplified. Furthermore, we propose extensions to current optimizers that decide on the exact ordering of flow tasks taking into account the new optimization metric. Finally, we evaluate the new optimization algorithms and show up to 59% response time improvement over state-of-the-art task ordering techniques.

Keywords Data flow optimization · Cost modeling · Task ordering

✉ Anastasios Gounaris
gounaria@csd.auth.gr

Georgia Kougka
georkoug@csd.auth.gr

1 Department of Informatics, Aristotle University of Thessaloniki, Thessaloníki, Greece

1 Introduction

Nowadays, data flows constitute an integral part of data analysis. The modern data flows are complex and executed in parallel systems, such as multi-core machines or clusters employing a wide range of diverse platforms like Pentaho Kettle,[1] Spark[2] and Flink[3] to name a few. These platforms operate in a manner that involves significant time overlapping and interplay between the constituent tasks in a flow. However, there are no cost models that provide analytic formulas for estimating the *response time (wall-clock time)* of a flow in such platforms. Cost models, apart from being useful in their own right, are encapsulated in cost-based optimizers; currently, for example, cost-based optimization solutions for task ordering in data flows employ simple cost models that may not capture the flow execution running time accurately, as shown in this work. For example, the sum cost metric, which is employed by many state-of-the-art task ordering techniques [17,20,30], merely sums the cost of individual tasks. This results in an execution cost computation that may deviate from the real execution time, and the corresponding optimizations may not be reflected on response time. Consequently, there is a need for employing new data flow optimization solutions that take into consideration a cost model during the optimization decision phase that is tailored to response time minimization.

Typically, cost models rely on the existence of appropriate metadata regarding each task, which are combined using simple algebraic formulas with the sum and max operations. Most often, task metadata consider the cost of each task, which is commensurate with the task running time if executed in a stand-alone manner. The main challenges in devising a cost model for running time that is appropriate for modern data flow execution stem from the following factors: (i) many tasks are executed in parallel employing all three main forms of parallelism, namely, partitioned, pipelined and independent, and the resulting time overlaps, which entail that certain task executions do not contribute to the overall running time, need to be reflected in the cost model; and (ii) computation resources are shared among multiple tasks, and the concurrent execution of tasks using the same resource pool impacts on their execution costs.

In this work, we initially focus on a single multi-core machine environment, such as Pentaho Data Integration (PDI, aka Kettle). First, we devise a cost model that can be used to estimate the response time, when the data flows are executed in parallel and distributed execution environments. Then, we show the inadequacy of the existing task ordering optimization algorithms and propose a new optimization algorithm that decides the best order of executing the tasks of a flow employing the proposed cost model. Regarding the proposed model, we build upon existing cost modeling techniques that tend to consider time overlapping (e.g., [1,2,6,23,31,33]), but not the interplay between task costs. In order to achieve this, we propose a solution in which the cost of each task is weighted according to the number of concurrent tasks taking into account constraints of execution machines, such as capacity in terms of number of cores. Then, we show cases, in which the existing optimization solutions fail to

[1] http://community.pentaho.com/projects/data-integration.

[2] http://spark.apache.org/.

[3] http://flink.apache/.

improve the response time of a flow execution and, to ameliorate this, we introduce new optimization techniques that utilize the more advanced cost model and build upon the effective combination of optimization techniques in [20,33]. The results of the optimization improvements are thoroughly validated. More specifically, the contribution is as follows:[4]

1. We explain and provide experimental evidence on why the existing cost models provide estimates that may widely deviate from the real execution time of modern workflows.
2. We propose a model that not only considers overlapping task executions but also quantifies the correlation between task costs due to concurrent allocation to the same processing unit. The model is execution engine software- and data flow type-independent.
3. We show how our model applies to example flows in PDI, where inaccuracies of up to 50% are observed if the impact of concurrency is not considered.
4. We explain why the state-of-the-art optimization algorithms may fail to optimize the response time of a data flow execution.
5. We propose new task ordering optimization techniques that leverage the cost model to decrease running time.
6. We conduct a thorough evaluation of the new optimization techniques and the results show that improvements can reach 59% over state-of-the art techniques that aim at minimizing response time indirectly, through minimizing resource consumption (sum cost metric). The average improvements over a group of random initial valid flows can be up to 4.87 times for flows with 24 tasks.

In the remainder of this section we provide background on flow parallelization, the assumptions regarding the execution environment that we consider and a discussion about the inadequacy of cost models employed in data flow optimization. We continue the discussion of related work in Sect. 2. In Sect. 3 we introduce the notation. Our modeling proposal is presented in detail in Sect. 4, while the in Sect. 5, we introduce the new task ordering optimization solutions. In Sect. 6, we show experimental evaluation findings based on extensive simulation. We discuss the barriers and challenges of applying and incorporating the proposed optimization techniques in tools such as PDI in Sect. 7. We summarize the conclusions in Sect. 8.

1.1 Parallelizing data flows

The parallel execution of a data flow exploits three types of parallelism, namely *inter-operator (pipelined)*, *intra-operator (partitioned)* and *independent* parallelism. Here, we use the terms *task* and *operator* interchangeably. These types of parallelization are well-known in query optimization [9], and used to decrease the *response time* of a data flow execution.

The *intra-operation parallelism* considers the parallelization of a single task of a data flow. This type of parallelization is defined by the instantiation of a logical task as

[4] An early version of this work containing the modeling material but no optimization solutions has appeared in [19]. The novel material in this work compared to the preliminary version is in Sects. 5, 6 and 7, while Sect. 2 has been also extended accordingly.

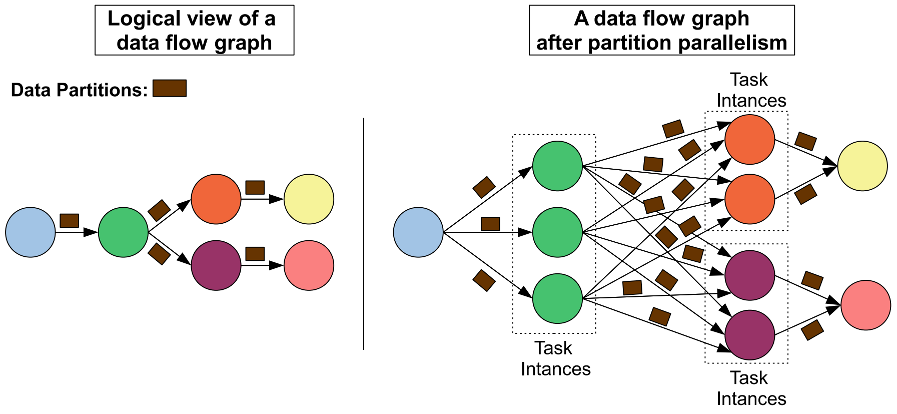

Logical view of a data flow graph
Data Partitions:
A data flow graph after partition parallelism
Task Intances
Task Intances
Task Intances

Fig. 1 A data flow graph before and after partitioned parallelism; circles with the same color correspond to partitioned instances of the same flow task (Color figure online)

a set of multiple physical instances, each operating on a different portion of data, i.e. each task can be executed by several processors and data is partitioned. An example of partitioned parallelism is depicted in Fig. 1. There is a set of different methods of partitioning, such as round-robin and hash-partitioning. In this work, we assume that the degree of *intra-operation* parallelism is fixed; e.g., in Fig. 1, the degree for the green task is set to 3.

The *independent* parallelism is achieved when the tasks of the same data flow may be executed in parallel because there are no dependency constraints or communication between them. An example is the two branches at the right part of the flow in Fig. 1(left).

The *pipeline* parallelism takes place when multiple tasks are executed in parallel with a producer-consumer link and each producer sends part of its output, which is a collection of records, as soon as this output is produced without waiting the processing of its input to complete and, therefore, the whole output to be produced.

In this work, we present a cost model for data flow execution plans that accurately estimates the *response time* considering the *pipeline parallelism* and *independent* types of parallelism, which are relevant to a single machine PDI execution. However, it is straightforward to extend our work to cover partitioned parallelism as well, as briefly discussed in Sect. 4.

1.2 Assumptions regarding a single multi-core machine execution environment

Our main assumptions are summarized as follows:

- Data flows utilize all the available machine cores. The number of cores depends on the execution machine.
- The execution machine is exclusively dedicated to the data flow execution. I.e., we assume that an execution machine executes only one data flow and the execution of the next flow can be started only after the completion of the previous flow. So,

the available machine executes tasks and stores data for a single data flow at a time.
- Multiple tasks of a data flow are executed simultaneously through multi-threading that allows multiple threads to be employed during flow processing. More specifically, we assume a form of multi-threaded execution, in which each task spawns a separate thread, running on a core decided by the underlying operating system scheduler. Obviously, if two task threads share the same core, they are executed concurrently but not simultaneously.
- The execution engine exploits pipeline and independent parallelism to the largest possible extent; i.e., the default engine configuration regarding task execution operates in a mode, according to which flow tasks are aggressively scheduled as soon as their input data is available.

The assumptions above hold also for massive parallel settings. The main difference is that, in massive parallelism settings, partitioned parallelism typically applies.

1.3 Motivation for devising a new cost model for task ordering optimization

A main application of cost models is in cost-based optimization. One of forms of data flow optimization that has been largely explored in the data management literature is task re-ordering. Taking this type of optimization as our case study, we can observe from the survey in [20] that the corresponding techniques target one of the following optimization objectives:

1. *Sum Cost Metric of the Full plan (SCM-F)*: minimize the sum of the task and communication costs of a data flow [13,16,17,22,28,30,31,36].
2. *Sum Cost Metric of the Critical Path (SCM-CP)*: minimize the sum of the task and communication costs along the flow's critical path [1,2].
3. *Bottleneck*: minimize the highest task cost [1,2,31,33].
4. *Throughput*: maximize the throughput (number of records processed per time unit) [8].

The first three metrics and the associated cost models can capture the *response time* under specific assumptions only. The *response time* represents the wall-clock time from the beginning until the end of the flow execution. *SCM-F* defines the *response time* when the tasks of a data flow are executed sequentially; for example when all tasks are blocking. Another case is when tasks are pipelined but are executed on the same CPU core (processor). In that case, the *SCM-F* may serve as a good approximation of the *response time*. *SCM-CP* reflects the *response time* when the data flow branches are executed independently and the tasks of each branch are executed sequentially. Finally, *bottleneck* represents the *response time* when all the tasks of the flow are executed in a pipelined manner and each task is executed on a different processor assuming enough cores are available.

So, why do we need another cost model? PDI, Flink, Spark and similar environments aggressively employ pipeline parallelism potentially on multiple processors. Consequently, the *SCM-F* and *SCM-CP* cost metrics do not correspond to the *response time* of the flow execution. In general, *SCM-F* indirectly aims to capture resource consumption; since the modern execution engines are advanced enough to perform sophisticated resource usage and avoid resource under-utilization, minimizing the resource consumption may be correlated to minimizing the response time in some cases on a single multi-core machine, but as shown later, this does not always hold and moreover, it is rare in a setting employing multiple machines. The *bottleneck* cost metric is not appropriate either. This is because there are pipelined tasks that are executed on the same processor, but also there are tasks that are blocking, e.g., sort. So, for estimating *response time*, we need to employ a new cost metric that explicitly considers parallelism and the corresponding overlaps in task execution.

Furthermore, a more accurate cost model for describing the response time is significant in its own right even when not used to drive optimizations. It allows us to better understand the flow execution and provides better insights into the details involved. Moreover, as will be shown in the subsequent section, merely considering time overlaps does not suffice, because the task costs are correlated during concurrent task execution.

2 Related work

We split the related work into two parts to discuss cost models and task re-ordering, respectively.

Cost modeling. The main limitation of existing cost models is that, even if they consider overlapped execution, they assume that the cost of each task remains fixed independently of whether other tasks are executing concurrently sharing CPU, memory and other resources. Examples that fall in this category are the work in [23], which targets a cloud environment for scientific workflow execution, and in [6]. The cost model in the latter considers that the flow is represented by a graph with multiple branches (or paths), where the tasks in each path are executed sequentially and multiple branches are executed in parallel. In contrast, we cover more generic cases.

Additionally, several proposals based on the traditional cost models have been presented in order to capture the execution of MapReduce jobs. For example, a performance model that estimates the job completion time is presented for ARIA Framework in [35]; this solution accounts for the fact that the map and reduce phases are executed sequentially employing partitioned parallelism but do not take into account the effect of allocation of multiple map/reduce tasks on the same core. The same rationale is also adapted by cost models introduced in proposals, such as [37] and [32]. Nevertheless, an interesting feature of these models is that they model the real-world phenomenon of imbalanced task partition running times. In the MapReduce setting, the authors in [29] propose the Produce-Transporter-Consumer model to define the parallel execution of MapReduce tasks. The key idea is to provide a cost model that describes the tradeoffs of four factors (namely, map and reduce waves, output compression of map tasks and copy speed during data shuffling) considering any overlaps. As previously, the impact of concurrency is neglected. Other works for MapReduce, such as [3], suffer from the same limitations.

Task ordering. Data flow optimization is a multi-dimensional area; broadly, optimizations are divided into two main categories, those referring to the logical (or conceptual) level and those referring to the low-level physical execution layer [20]. In the first category, the corresponding techniques employ task ordering, introduction, removal, merge and decomposition. In the latter, they focus on choosing the task implementation, execution engine and its configuration among several alternatives. In this work, we focus on task ordering, which refers to the logical level of the flow description but requires an underlying cost model that considers the low-level physical execution details.

A significant number of techniques that optimize the flow execution plan through changes in the structure of the flow graph including task ordering mechanism have been presented in the literature. The key characteristic of these proposals is that they consider other cost models than response time during optimization decisions, such as bottleneck and sum cost metric. For example, there are flow optimization solutions that are inspired by query processing techniques. In [10], an optimization algorithm for query plans with dependency constraints between algebraic operators is presented. The techniques in [17] build on top of [14,21], and are shown to be superior to approaches, such as [5,12,15,22,26,30,36] when *SCM-F* is targeted. In [18], an exhaustive optimization proposal for data flows is presented that aims to produce all the topological sortings of the tasks in a way that each sorting is produced from the previous one with the minimal amount of changes. This technique can be adapted for minimizing the response time, but is not scalable for data flows with high number of tasks and especially, for flows that preserve only a few dependency constraints between tasks. An early idea of metric combination was presented in [31], where the pipelining segments are grouped and for these sub-flows, the *response time* equals to the *bottleneck* metric. Then *SCM-F* is minimized for these segments of pipelining tasks. However, the pipelined segments are not optimized and independent parallelism is not considered.

Another interesting approach to flow optimization is presented in [13], where the optimizations are based on the analysis of the properties of user-defined functions that implement the data processing logic. This work focuses mostly on techniques that infer the dependency constraints between tasks through examination of their internal semantics rather than on task re-ordering algorithms per se. In general, automated extraction of statistical and semantic task metadata is of key significance in order task re-ordering techniques to find their way into business data flow execution platforms.

3 Preliminaries

A data flow is represented as a *Directed Acyclic Graph (DAG)*, where each vertex corresponds to a task of the flow and the edges between vertices represent the communication among tasks (intermediate data shipping among tasks). In data flows, the exchange of data between tasks is explicitly represented through edges. We assume that the data flows can have multiple *sources* and multiple *sinks*. A *source* of a data flow corresponds to a task with no incoming edges, while a *sink* corresponds to a task with no outgoing edges. The main notation and assumptions are as follows:

Let $G = (V, E)$ be a *Directed Acyclic Graph (DAG)*, where $V = t_1, t_2, \ldots, t_n$ denotes the vertices of the graph (data flow tasks) and E represents the edges (flow of data among the tasks); n is the total number of vertices. Each vertex corresponds to a data flow task and is responsible for one or both of the following: (i) reading or storing data, and (ii) manipulating data. The tasks of a data flow may be complex data analysis tasks, but may also execute traditional relational operations, such as union and join. Each edge equals to an ordered pair (v_j, v_k), which means that task t_j sends data to task t_k.

Each data flow is characterized by the following metadata:

- *Cost* (c_i), which applies to each task. The c_i corresponds to the cost for processing all the input records that the t_i task receives taking into consideration the required CPU cycles and disk I/Os. In distributed systems, the cost of network traffic needs to be considered as well, and may be the most important factor. An essentially similar consideration is c_i to denote the cost per single input record. In the latter case, the *selectivity (sel)* information of all tasks is needed in order to derive the size of the task input and then, the task cost for its entire input; the selectivity denotes the average number of returned data items per source tuple.
- *Communication Cost* $(cc_{i \rightarrow j})$, which may apply to edges. The communication cost of data shipping between the t_i and t_j depends on either the forward local pipelined data transfer between tasks or the data shuffling between parallel instances of the same data flow. It does not include any communication-related cost included in c_i; it includes only the cost that is dependent on both t_i and t_j rather than only on t_i.
- *Parallelism Type of Task* (pt_i), which describes the type of parallelism of a task i, when the task is executed. More specifically, the parallelism type characterizes if a data flow task is executed in a pipelined, denoted as p or no pipelined manner (*blocking* task), denoted as np. A *blocking* task requires all the tuples of the input data in order to start producing results; i.e., the parallelism type of a task reflects the way a task process the input data and produces its output.

Note that the modeling of the flow as described above is independent of the input data types, which can be relational records, semi-structured documents or plain text items.

4 A cost model for data flow response time

First, we describe a model for a single-machine setting and finally, we generalize to distributed settings.

4.1 Models for a single multi-core machine

We start by examining simple flows and we gradually extend our observations to larger and more complicated ones.

4.1.1 A linear flow with a single pipelined segment of *n* tasks

A pipeline segment is defined by a sequence of n tasks in a chain, where the first task is either a source or a child of a blocking task. The last task is either a sink or a blocking task; both types of tasks do not allow flow of output records in a pipelined manner downstream. Additionally, the tasks in between are all of p type. Also, pipeline segments do not overlap with regards to the vertices they cover. Such segments benefit from inter-operator parallelism. The key point of our approach is to account for the fact that there is non-negligible interference between tasks. This interference is captured by the variable α. Let us suppose that our machine has m cores. In the case where $n \leq m$, each task thread can execute on a separate core exclusively. The cost model that estimates the *response time (RT)* of a data flow execution is defined as follows, which aims to capture the fact that the running times of tasks overlap.

$$Response\ Time\ (RT) = \alpha \max\{c_1, \ldots, c_n\} \tag{1}$$

In general, the parameter α is a weight that aims to abstract the impact of multi-threading in a single metric. Multi-threading may lead to performance overhead due to several factors, such as the *context switching* between threads, as the flow tasks are executed concurrently and need to switch from one thread to another multiple times. An additional factor for response time increase is due to the *locks* that temporarily restrict tasks sharing memory to write to the same memory location. Finally, the most significant factor in the terms of affecting the response time is the *contention* that captures the interference of the multiple interactions of each data flow task with memory and disk. Specifically, when there are multiple requests to memory, this may result in exceeding memory bandwidth and consequently, to RT increase. Finally, allocating and scheduling threads incurs some overhead, which, however, is negligible in most cases. Instead of devising complex cost models for all the above factors, in this work, we have decided to cover all of them by using a single parameter, which can be computed empirically through experiments.

Nevertheless, multi-threading execution leads to execution cost improvement because of the parallel task execution. So, we may observe RT minimization, when all or more of the available cores are exploited by the data flow tasks and one copy of data is used by multiple threads at the same time. Also, the delays occurred by transferring data from memory and disk are overlapped by the task execution, when the number of tasks is higher than the available execution units. In general, cache-level configuration may heavily impact on RT. As previously, all these factors are reflected on the α parameter.

Let us consider now the case where $n > m$ and the task threads need to share the available cores in order to be executed. In this case, each core may execute more than one task and the RT is determined by all the flow tasks. An exception is when there is a single task with cost higher than the sum of all the other costs (similarly to the modeling in [35]):

$$Response\ Time\ (RT) = \alpha \max\{\max\{c_1, .., c_n\}, \frac{\sum\{c_1, \ldots, c_n\}}{m}\} \tag{2}$$

4.1.2 Experiments in PDI

In the following, we present a set of experiments that we conducted in order to understand the role of α in RT estimation according to Eqs. (1) and (2). We consider synthetic flows in PDI with $n = 1, \ldots, 26$ tasks and an additional source task. The input ranges from 2.4M to 21.8M records. Two machines are used, with (i) a 4-core/4-thread i5 processor; and (ii) a 4-core/8-thread i7 processor, respectively. Finally, the task types are two, either homogenous or heterogeneous. In the former case, all tasks have the same cost (denoted as *equal*). In the latter case (denoted as *mixed*), half of the tasks have the same cost as in the *equal* case, and the other tasks have another cost, which is lower by an order of magnitude. All the tasks apply filters to the input data, but these filters are not selective in the sense that they produce the same data that they receive; they just incur processing cost. The data input is according to the TPC-DI Benchmark[25] and we consider records taken from the implementation in http://www.essi.upc.edu/dtim/blog/post/tpc-di-etls-using-pdi-aka-kettle. Each experiment run was repeated 5 times and the median times are reported; in all experiments the standard deviation was negligible.

The left column of Figs. 2 and 3 shows how the response time of the two different types of data flows evolves as the number of tasks, and consequently the number of execution threads, increases. It also shows what the cost model estimates would be if no weights were considered. The main observation is twofold. First, the response time, as expected from Eqs. (1) and (2), stays approximately stable when $n \leq m$, and then, grows linearly when $n > m$. This behavior does not change with the increase in the data size. Second, estimates with no weights can underestimate the running time by up to 50%, whereas there are also cases when they overestimate the running times by a smaller factor (approx. 5%). More importantly, the main inaccuracies are observed in the mixed-cost case, which is more common in practice.

The α factor is shown in the right column of Figs. 2 and 3. Values both lower and higher than 1 are observed. Although α captures the combination of overhead and improvement causes described in the previous section, the importance of each cause varies. In values greater than 1, resource contention is dominating; whereas, in values lower than 1, the fact that waits for resources are hidden outweighs any overheads. The main observations are as follows: (i) the α factor varies significantly for the same dataset when the number of tasks is modified; (ii) α can be of significant magnitude corresponding to more than 50% increase in the task costs; (iii) for flows that consist of up to 4 tasks with equal cost, the α factor continuously grows (i.e., contention is dominating) and then, when the number of tasks further increases, the behavior differs between cases; and (iv) for data flows with different task costs and $n > m$, the α factor increases sharply for flows with up to 7-9 tasks depending on the input data size.

4.1.3 A linear flow with multiple independent pipelined segments

In Table 1, we show the running times of flows with the same number of tasks when all tasks belong to a single pipelined segment and when there are two segments belonging to two different branches originating from the same source, according to the fork template in [34]. We can observe that the running times are similar. From this observation, we can draw the conclusion that the magnitude of the weights (i.e., the w^c and the corresponding α factors) depend on the number of concurrent tasks and need not be segment-specific; that is, it is safely to assume that all concurrent tasks share the same factors.

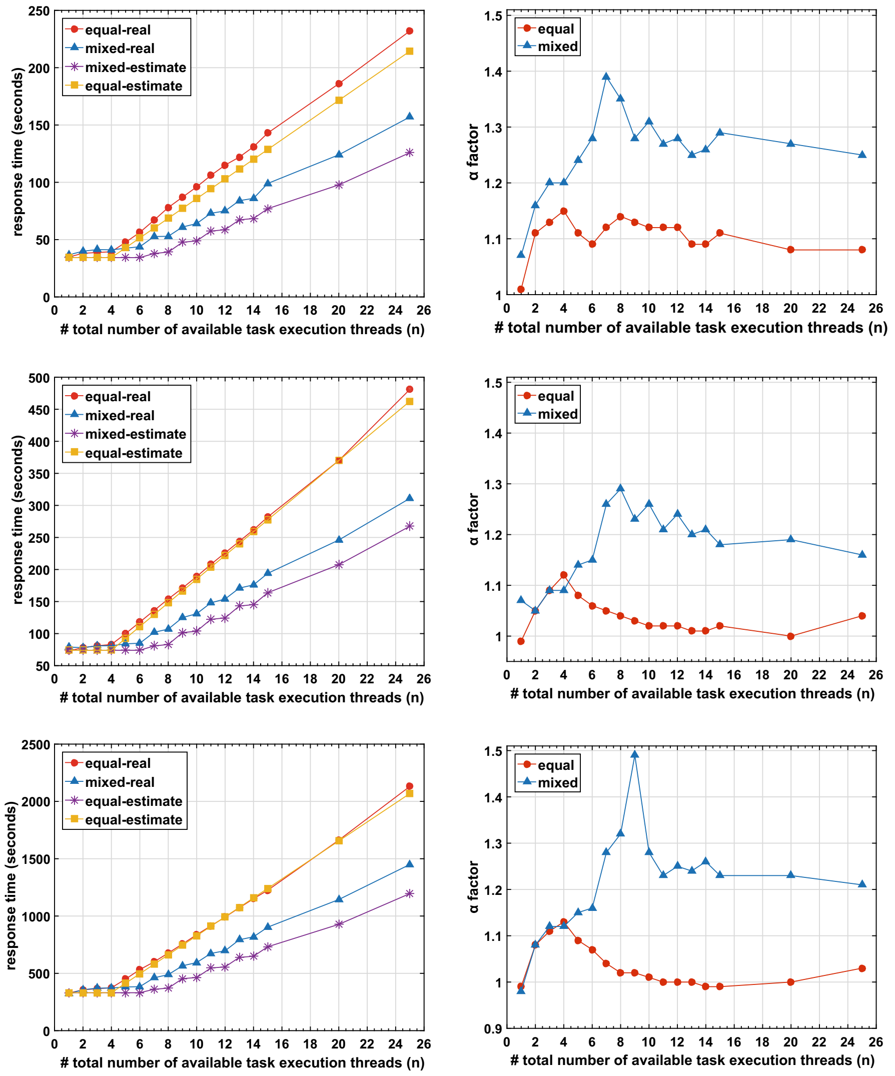


Fig. 2 Response Time (RT) and the α factor of linear flows with same and different task costs for $n \in [1, 25]$ executed by the 4-core/4-thread i5 machine for 2.4 (top), 4.8 (middle) and 21.8 M (bottom) input records

4.1.4 Estimating the response time of a flow: the complete case

In the previous sections, we showed how we can estimate the response time of a single pipelined segment in data flows. Now, we leverage our proposal to more generic data flows with multiple pipeline segments, in order to estimate the response time of flows that consist of multiple pipeline segments. To this end, we employ a simple list scheduling simulator. The steps of this methodology are described, as follows:

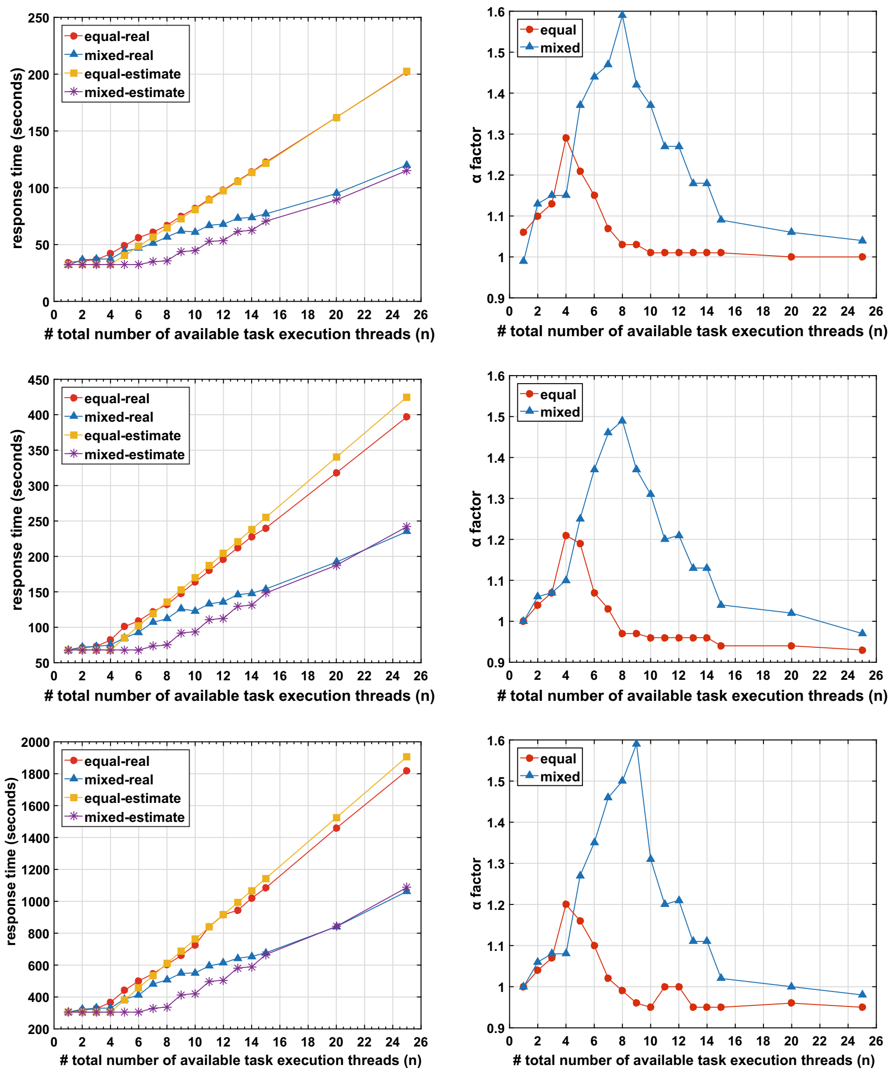


Fig. 3 Response Time (RT) and the α factor of linear flows with same and different task costs for $n \in [1, 25]$ executed by the 4-core/8-thread i7 machine for 2.4 (top), 4.8 (middle) and 21.8 M (bottom) input records

1. Receive as input the flow DAG, the cost (c_i) of all the tasks of a dataflow, the number of available cores, and the α factors.
2. Isolate all the single-pipeline segments of the flow with the help of the parallelism type task metadata.
3. Split the input in blocks of a fixed size B.
4. Create a copy for the first block for each task directly connected to a source and insert it in a FCFS (First Come First Serve) queue.

Table 1 Comparison of running times between flows with the same number of tasks but a) with 2 independent and b) a single segment (in seconds)

	4 Cores-4 threads		4 Cores-8 threads	
n	2 Branches	1 Branch	2 Branches	1 Branch
2.4 million records				
2	38.5	38.1	35.1	35.6
4	42	39.4	42.1	41.9
6	59.3	56.3	56	56.1
8	78	78	67	67
10	96	96	83	82
12	115	115	99	98
14	134	131	115	114
16	153	152	131	129
18	164	170	145	145
20	187	186	160	162
4.8 million records				
2	78	78	69	71
4	84	83	82	82
6	118	118	109	109
8	155	154	134	132
10	192	189	166	164
12	229	226	197	196
14	265	262	229	228
16	304	305	260	256
18	327	323	287	289
20	380	370	323	318
21.8 million records				
2	346	356	315	317
4	369	374	370	377
6	530	533	497	499
8	694	677	606	594
10	851	837	735	727
12	1040	991	879	916
14	1210	1151	1006	1019
16	1382	1311	1146	1152
18	1486	1493	1296	1299
20	1720	1662	1454	1437

5. Schedule blocks arbitrarily to cores until there are no blocks in the queue under the condition that a task can process at most one block at time at any core:
   (a) when a block finishes its execution, re-insert it in the queue annotated by the subsequent tasks in the DAG;
   (b) if the task is a child of a source or a block operator, insert its next block in the queue ;

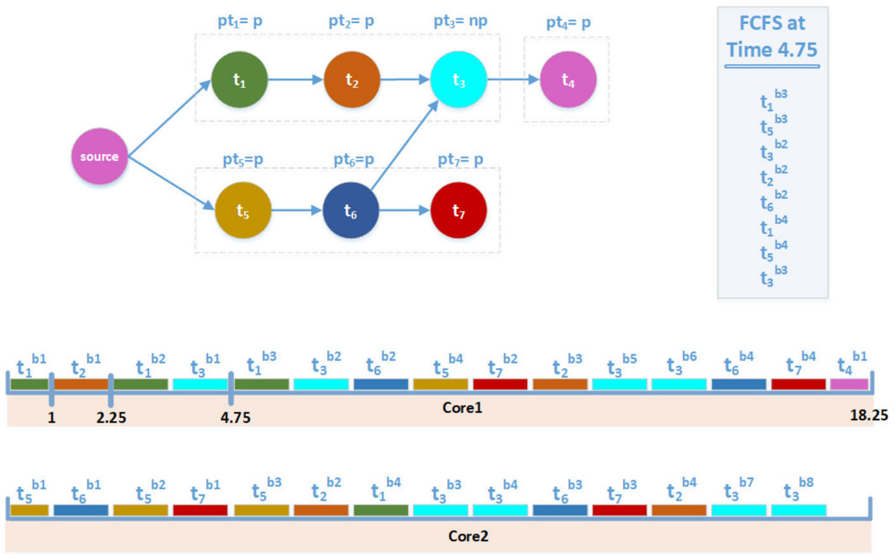


Fig. 4 Example of running a generic flow on 2 cores; the dotted borders denote pipeline segments

(c) if a blocking task has received its entire input, start scheduling the corresponding blocks for the segment initiating from this task.

6. The response time is defined by the longest sequence of block allocations to a core.

In Fig. 4, we present an example with a flow running on 2 cores, where all task costs per block are 1, each task receives as input 4 blocks and emits 4 other processed blocks except t_3, which outputs a single block. The α factor is 1, when there are up to two concurrent tasks, and 1.25 otherwise. Concurrent tasks are those for which there is at least 1 block either in the queue or being executed. In this example, the running time is when the work on the first is completed.

The methodology above assumes knowledge about the α factors. This knowledge can be derived through experiments similar to those in the previous sections (see also the discussion in Sect. 7).

4.2 A generalized cost model for response time

The cost models discussed previously can be seen as an instantiation of a more general model. More specifically, we define the following generalized cost model for estimating the response time:

$$Response\ Time\ (RT) = \sum z_i w^c c_i + \sum z_{ij} w^{cc} cc_{i \to j} \tag{3}$$

where variable $z_i = \{0, 1\}$ is binary and defined as 1 only for tasks that determine the RT. Similarly, $z_{ij} = \{0, 1\}$ is set to 1, only when $cc_{i \to j}$ contributes to the total RT. The c_i factor denotes the cost of the i^{th} task, where $i = \{1, \ldots, n\}$. The w^c and w^{cc} weights generalize the α parameter and thus cover a set of different factors that are responsible for the increase/decrease of RT during the task execution and communication between two tasks (data shipping), respectively. In a nutshell, the z variables capture the time overlapping of different tasks, whereas w^c and w^{cc} quantify the impact of the execution of one task on all the other tasks that are concurrently executed, i.e., they capture the correlation between the execution of multiple concurrent tasks.

Eq. (3) is reduced to Eq. (1) if we set (i) $z_i = 0$ for all tasks, apart from the task with the maximum cost, for which z is set to 1, since it determines the RT; (ii) $cc_{i \to j}$ is set to 0 for all pairs (i, j) and (iii) $w^c = \alpha$: To derive Eq. (2), w^c equals either to α, as in Eq. (1) or to α/m with $z_i = 1$ for all the flow tasks.

More importantly, the cost model in Eq. (3) generalizes the traditional ones discussed in Sect. 1.3. For example, based on the proposed formula, if we consider w^c and w^{cc} set to 1 and that all tasks have $z_i = 1$, then the cost model actually becomes equivalent to *SCM-F*. If only the tasks that belong to the critical path have $z_i = 1$, and we keep w^c and w^{cc} set to 1, then the cost model corresponds to *SCM-CP*. Similarly, if we want to consider the bottleneck cost metric, we can set $z_i = 1$ in Eq. (3) for the most expensive task and $z_i = 0$ for all the other tasks.

4.2.1 Considering communication costs

We need to consider communication only in settings where multiple machines are employed. Broadly, we can distinguish among the following three cases:

1. On each sender, there is a single thread for computation and transmission. In this case, both z_i and $z_{i,j}$ in Eq. (3) are 1 to denote that computation and transmission occur sequentially.
2. On each sender, there is a separate thread for data transmission, regardless of the number of outgoing edges. In this case, depending on which type of cost dominates, only one of z_i and $z_{i,j}$ is set to 1, since computation and transmission overlap in time.
3. On each sender or receiver, there is a separate thread for each edge. If all edges share the same network, then we can follow the same approach as in the case of multiple pipelined tasks sharing a single core.

The first two cases assume a push based data communication model, whereas the third one applies to both push and pull models.

4.2.2 Considering partitioned parallelism

Partitioned flows running on multiple machines can be covered by our model as well. More specifically, we can model and estimate the DAG flow instance on each machine independently using the same approach, and then take the maximum running time as the final one. The factors may differ between partitioned tasks. Finally, if a DAG instance does not start its execution immediately, we need to add the time to receive its first input (which kicks-off its execution) to its estimated running time.

5 Optimizing a data flow for response time

Informally, the problem of optimizing task ordering is to define a partial order of tasks, so that dependency constraints are respected and a given objective function (e.g., $SCM - F$, *bottleneck*, and so on) is minimized. A more formal definition can be given for optimizing linear flows, after introducing the notion of precedence constraints, as follows.

Definition 1 Let $PC = (V, D)$ be another DAG capturing the precedence constraints of a linear flow $G_{linear}(V, E)$. D defines the precedence constraints (dependencies) that might exist between pairs of tasks in V: $D = \{d_1, \ldots, d_l\}$ is a set of l ordered pairs: $d_i = (t_j, t_k), 1 \leq i \leq l,\ 1 \leq j < k \leq n$, where each such pair denotes that t_j must precede t_k in any valid G.

The above definition states that an optimized G should contain a path from t_j to $t_k,\ \forall(t_j, t_k) \in D$. Essentially, the PC graph defines constraints on the valid edges of the G graph, where G is linear. This also implies that if D contains (t_a, t_b) and (t_b, t_c), it must also contain (t_a, t_c). Note that the PC and G graphs are semantically different, as the PC graph corresponds to a higher-level, non-executable view of a data flow, where the exact ordering of tasks is not defined; only a partial ordering is defined instead. With the help of PC, we can define our task ordering optimization problem of linear G flows.

Definition 2 Our task ordering optimization problem is defined as follows: Given a set of tasks V, $PC(V, D)$ and the cost per input record c_i and selectivity sel_i metadata for each task $t_i \in V$, find a total ordering of V that minimizes RT.

Solving the above problem finds an optimized ordering of tasks only regarding a single flow branch in generic data flow structures, since it refers to linear flows. Our rationale is to split complex flows into their linear components (see [17] for a discussion), and optimize such linear sub-flows individually.

Even when narrowing our focus on linear flows, the problem is NP-hard. As explained in Sect. 4, minimizing RT generalizes the problem of minimizing $SCM - F$. However, the latter is intractable, and moreover, *"it is unlikely that any polynomial time algorithm can approximate the optimal plan to within a factor of* $O(n^\theta)$", where θ is some positive constant, as explained in [4].

As already discussed in Sect. 1.3, there are several techniques that already optimize flows but considering other optimization criteria. Nevertheless, their rationale is useful in order to build our solution. The two key aspects, upon which we build to avoid re-inventing the wheel, are:

1. To optimize $SCM - F$, ordering by the rank value yields better solutions. Rank is defined as $rank(t_i) = \frac{1-sel(t_i)}{c_i}$, where c_i refers to the cost of t_i per input record. Such an approach is followed by [17], which further elaborates on how precedence constraints are satisfied in a scalable manner.
2. To optimize *bottleneck*, first placing the selective tasks followed by the non-selective tasks and then ordering the selective tasks by their cost in ascending order yields better solutions. Such an approach is followed by [33].

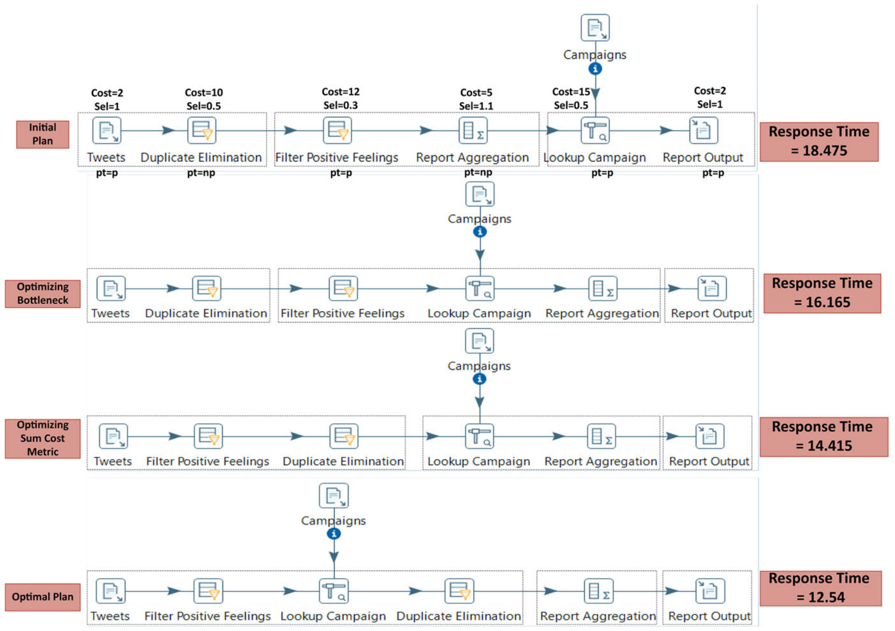


Fig. 5 Optimizing a real data flow processing Twitter records.Top: initial plan. Middle: two plans optimized according to current state-of-the-art techniques. Bottom: our solution

We leverage both these techniques in order to devise a solution that targets *RT*. However, before explaining our solution, we explain their inadequacy through an example.

5.1 Inadequacy of existing techniques: an example

Figure 5 depicts a simple scenario of a real linear data flow, two optimized plans of the flow using two different optimization techniques that minimize the bottleneck and the sum of all the task costs, respectively, and the optimal execution order of this data flow. More specifically, this linear data flow consists of six tasks (or processes) that form a chain, i.e., each non-source and non-sink task has exactly one incoming and one outgoing edge. The flow aims to process text data retrieved from *Twitter* (tweets) that comment products in order to produce a dynamic report related to the products that users prefer and advertisement campaigns. Each of the tasks is annotated with a selectivity and cost value that express the average number of returned data items (tweets) per source tuple for a flow task and the time cost of each task per record processed, respectively. In the case that a task has $sel < 0$, the corresponding task produces less output records for each input record, whereas, $sel > 0$ indicates the production of more output records than the input records. The first task, labeled as *Tweets*, is the task that feeds the flow with data, the following task (*Duplicate Elimination*) performs a sort-based duplicate elimination, while the *Filter Positive Feelings* task is a filter for extracting only the positive feelings of the Twitter records. Additionally, the

Report Aggregation is a task that produces reports, the *Lookup Campaigns* performs a lookup in a static data source to match each of the tweeted products with a set of related campaigns for each product and finally, the *Report Output* task produces the final report. Based on their semantics, *Duplicate Elimination* and *Report Aggregation* are blocking tasks.

In the figure, we depict the initial plan and the results of the task ordering optimization when we target minimizing the bottleneck cost (according to the technique in [33]), the sum cost metric (according to the techniques in [17]) and the response time (this work). On the right, the response time per input record in each case is shown, where we see that the technique presented hereby (bottom plan) is capable of yielding significant performance improvements. When calculating the response time, for simplicity, we assume that the α factor is set to 1. As such, RT is defined as the sum of the maximum task costs of each pipeline segment. The pipeline segments are delineated by the dotted lines in the figure. During optimization, all dependency constraints are respected; the constraints in this example define that *Duplicate Elimination* should always precede *Report Aggregation* and, trivially, the source and the sink tasks should not be moved.

The key lesson from this example is that when ordering the selective tasks by their costs as in [33] or by their rank value, as in [17], neglecting the allocation of tasks to pipeline segments, the overall response time may not be minimized.

5.2 Our solution

The basis of our solution is to split the initial flow into linear segments, each optimized individually. Algorithm 1 summarizes our approach. We first optimize the linear sub-flows using the best-performing technique in [17], called $RO - III$. This optimization is driven by the rank values, as explained above. Then, each linear sub-flow is decomposed into its pipelined segments.

For each such segment, we repeat a 3-phase process. In the first phase, inspired by the work in [33], we order the tasks in a way that the selective ones are moved upstream and their relative order is only by their cost. The `check-moving` operation in Algorithm 1 is responsible for (i) checking whether the move does not violate the constraints in PC; and (ii) running the simulator in Sect. 4 in order to establish as to whether the move is beneficial or not. In the second phase, the algorithm attempts to modify the pipelined segments in the same branch with a view to moving an expensive task of one segment to an adjacent segment, when the cost of the expensive tasks is going to be hidden due to overlapped execution. This type of optimization is exemplified in the bottom plan in Fig. 5. Since the second step may have modified the contents of the pipelined segments, in the third phase we repeat the process of the first phase.

Two main features of this solution are: (i) By its design, it cannot yield worse plans in terms of *RT* than *RO-III*; i.e., our solutions either retains the solution of *RO-III* or produces a faster plan. (ii) The optimization is machine-type specific, since it heavily relies on the simulator in Sect. 4, which is parameterized by the machine type-specific α factors (see also the algorithm input). This implies that it must be repeated for the same flow when the execution engine host is modified. In practice, the optimization

Algorithm 1 Task re-ordering for optimizing RT

```
Input: $G(V, E),\ PC(V, E),\ c_i, sel_i, pt_i\ \forall t_i,\ \alpha(p), p = 1 \dots |V|$

split flow into linear branches
for each branch $G_{linear}(V, E)$ and its corresponding $PC(V, D)$ do
    plan $P \leftarrow$ optimize $G_{linear}$ for $SCM - F$ using the $RO - III$ algorithm in [17]

    //Phase-I
    identify pipelined segment boundaries in $P$ (i.e., tasks with $pt = np$)
    for each pipelined segment $SP$ (sub-flow of $P$) do
        for i in 1 ...size(SP) do
            $t_{best} \leftarrow$ the task after position $i$ with $sel < 1$ and the lowest cost
            if $\exists t_{best}$ then
                check-moving $t_{best}$ to position $i$ and shift other tasks to the right

    //Phase-II
    for each pipelined segment $SP$ (sub-flow of $P$) do
        for i in 1 ...size(SP) do
            check-moving $t$ in position $i$ at the beginning of the next $SP$
    for each pipelined segment $SP$ (sub-flow of $P$) do
        for i in 1 ...size(SP) do
            check-moving $t$ in position $i$ at the end of the previous $SP$

    //Phase-III
    Repeat Phase-I
```

time is dominated by $RO - III$ and the list scheduler simulator, but is negligible in machines, such as those used in Sect. 4, i.e., it does not exceed a few seconds, even for flows of 24 tasks.

5.2.1 Analysis

Let a flow with n tasks and its largest linear sub-flow containing n_l tasks. Extracting linear sub-flows and applying *RO-III* to each such sub-flow takes $O(n)$ and $O(n_l^2)$ time, respectively [17]. `check-moving` is an $O(n^2)$ process regarding checking violations of the dependencies and $O(n)$ to estimate *RT* through simulation. In the worst case, the nested loop in Phase-I in Algorithm 1 is repeated n_l times. Therefore, the complexity of Phase-I is $O(n_l n^2) = O(n^3)$. Similarly, the complexity of the next two phases is also $O(n^3)$. Overall, our solution is of cubic complexity, which is practical for very large flows of 100-200 tasks.

6 Evaluation of the response time optimizations

In the initial set of experiments, we focus on linear flows and we conduct a thorough evaluation varying three dimensions: (i) the number of tasks in the flow; (ii) the percentage of the edges in $PC(V, D)$ compared to the case that D is a complete graph; and (iii) the percentage of blocking operators. The higher the percentage of PC edges is, the less flexibility in re-ordering tasks exists. Also, the higher the percentage of blocking operators, the higher the number of pipelined segments in the flow. For each

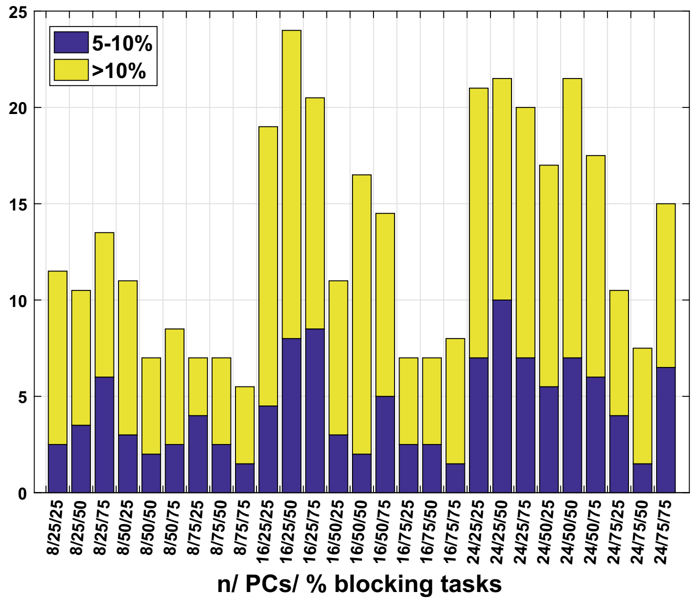


Fig. 6 Percentage of cases in which our solutions yielded lower RT than *RO-III* by at least 5%

dimension, we examined three values: 8, 16 and 24 tasks, and 25%, 50% and 75% for the two percentages. Overall, we investigated $3^3 = 27$ combinations. Each combination is simulated 200 times, with the a factor being the same as the one in Fig. 3(top). In each run, we assigned random values to the cost and selectivity task metadata; the cost ranged from 1 to 100, and the selectivity ranged from 0.01 to 1.5.

First, we compare the improvements upon the algorithm *RO-III* in [17]. As this algorithm is the best one to date in minimizing the sum cost metric, which is roughly equivalent to minimizing resource consumption, someone would expect to yield very good running times as well. Indeed, this is the case, i.e., in many tests, there was no improvement. However, there is a significant number of cases, in which RT can further drop. Fig. 6 shows the distribution of such numbers per each combination of the three dimensions, separately. We can observe that, for some combinations, the improvements are more frequent than in 20% of the cases, whereas in 2/3 of the combinations, at least 10% of the plans were improved. In the figure, we also distinguish between small (5-10%) and larger ($> 10\%$) RT improvements, and we totally omit improvements less than 5%, as non-significant. In general, as expected, there are more improvements when there are (i) fewer constraints, since there is more flexibility in re-ordering and moving tasks across pipelined segments, (ii) the flows are larger.

In Fig. 7, we show the maximum magnitude of reductions in RT observed, compared to the plan produced by *RO-III*. Such reductions can be higher than 59%, i.e., the plan derived by the proposed solutions runs more than 2 times faster.

Thus far, we have concentrated on how our solution improves upon the state-of-the-art. We now shift our focus on how significant the optimizations are in general, and we compare against initial valid data flows, which are randomly generated under

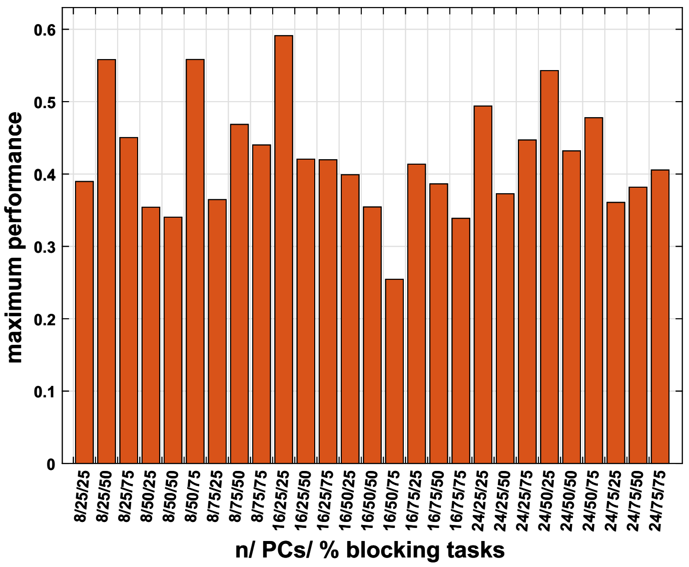


Fig. 7 Maximum reductions in RT compared to the plan of *RO-III*

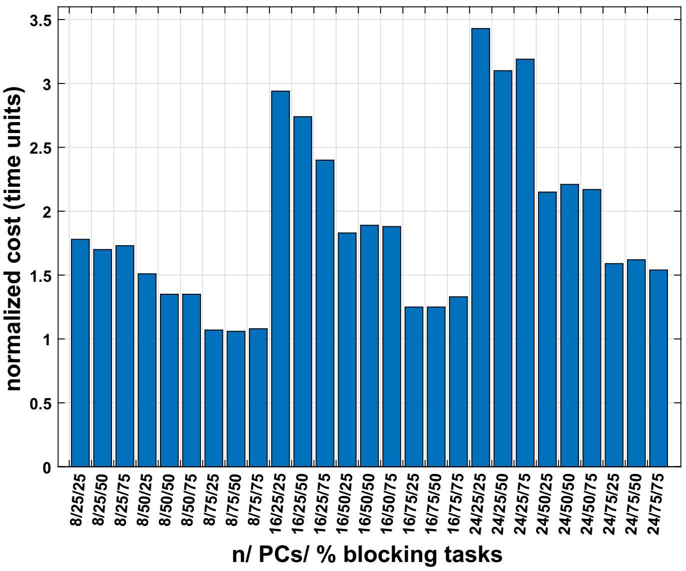
the single requirement to respect the dependencies. Figure 8 shows the median and average times such solutions are slower for each group of 200 runs. As previously, the size of the flow and the percentage of the constraints significantly impact on the behavior. The average values are, as expected, higher than the median ones, and can reach up to 4.87 times better performance improvement.

As a next step, we evaluate fork flows [34] with two branches and flows with higher maximum selectivity, i.e., up to 2. The main observations are twofold: firstly, the higher the number of non-selective tasks in the flow, the lower the number of optimizations, and secondly, it is less effective to apply our optimizations to a two-branch flow with n tasks than to a single-branch (linear) flow of $2n$ tasks. The latter holds because the optimizations in one branch may be reflected on response time only if this branch dominates the execution time.

Finally, in Fig. 9, we present optimization times when our algorithm runs on a i7 CPU at 2.6GHz with 16GB RAM, where we can see that for small flows, the optimization overhead is less than 1 sec, whereas, in the most complex cases, it remains below 15 secs even when there is significant flexibility in task re-ordering. These times include running *RO-III* at the beginning and the list scheduler simulator within the algorithm steps, whenever *RT* estimates are required.

7 Discussion on incorporating the solution into a real system

A question that naturally arises is: Are we now ready to incorporate the solution into a real system, such as PDI? Similarly, why we were not able to test the optimizer in real



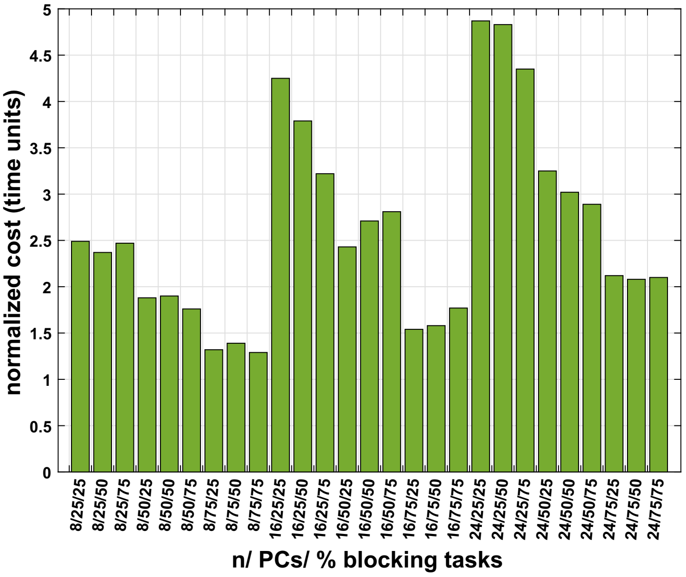


Fig. 8 Median (top) and average (bottom) times the initial data flow with n tasks, % PCs and blocking tasks is slower than the optimized one

flows, such as those of TPC-DI?[5] The answer to these questions is that the solution proposed in this work is a cost-based one, and, as such, it relies on the existence of the

[5] We were able to emulate flows in Kettle, where tasks had arbitrary cost and selectivity values through Javascript user-defined functions. However, for this specific type of transformation step, the execution of PDI is not parallel as usual, and the response time becomes more commensurate to the *SCM-F*. To avoid confusion of the readers, we omitted all these runs of synthetic flows in a real system.

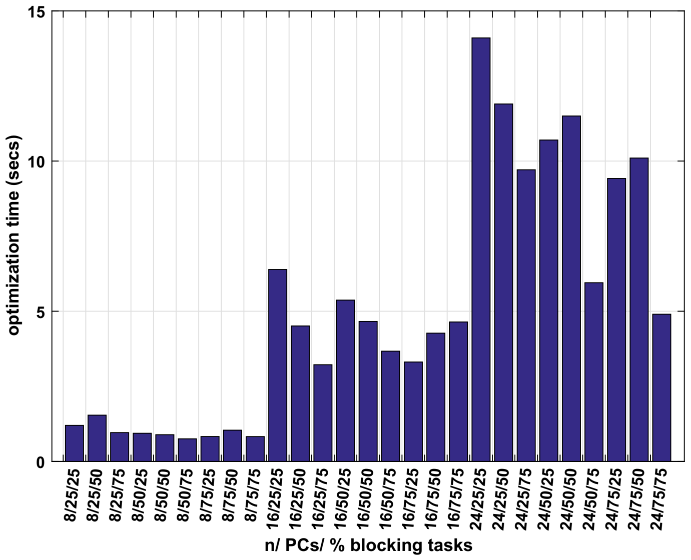


Fig. 9 Optimization time of full Algorithm 1

cost and selectivity metadata. How to reliably and efficiently extract such metadata is an open issue.

There is early work on statistics collection [7,11,24], but, a practical manner to automatically extract cost and selectivity metadata is still missing, given also that selectivity values are also typically correlated and task execution times that are typically reported encompass overheads of the complete flow execution. However, deriving statistical metadata using a combination of past logs and micro-benchmarking is a promising avenue for future research, so that data flow optimization techniques can be encapsulated in execution engines.

Another aspect is how to automatically extract the dependencies between tasks. Manually deriving them is a feasible yet tedious tasks. However, a complete solution should incorporate techniques for automatically detecting semantic constraints between tasks. This line of research has also produced interesting but not mature results yet [27].

Overall, due to recent advances in optimization algorithms for data flows, including the proposal in this work, the main barriers in incorporating the techniques in a real system have been shifted from the need to devise scalable algorithms handling constraints to the problem of deriving the required statistical and semantic input metadata. If the latter problem is addressed, then our method can be applied as a plug-in component to any flow execution engine that can export the flow structure. For example, we have successfully created such a plug-in for manually fed (and artificially created) metadata for both PDI and Camunda, a business process management tool for testing simple processes in BPMN 2.0 that can be mapped to DAGs. In both cases, we export the flow description in XML, we parse it and map it to an annotated DAG, where annotations correspond to metadata, we apply the optimizations and we transform it back to its native format.

8 Conclusions and future work

In this work, we address a limitation of existing data flow cost models that do not accurately estimate the *response time* of real data flow execution, which heavily depends on parallelism. We propose a model that considers the time overlaps during the task execution, while it is capable of quantifying the impact of concurrent task execution. The latter is an aspect largely overlooked to date and may lead to significant inaccuracies if neglected, e.g., we provided simple examples of deviations up to 50%. Additionally, we propose an optimization solution that aims to improve the response time of a data flow by defining the execution order of the flow tasks based on the proposed cost model. In our experiments, the proposed optimization technique has shown to yield improvements of up to 59% compared to the state-of-the-art in data flow task ordering.

Our work can be extended in two complementary ways. Firstly, to work towards end-to-end solutions with a view to incorporating the techniques in a real system, as discussed in the previous section. Secondly, applying the proposed model relies on the existence of accurate machine type-specific weight information; deriving efficient ways to approximate the weights before flow execution and generalize over types of execution engine hosts is an open issue. Finally, another direction for future work is to make a deep dive into the low-level resource utilization and wait measurements to establish the detailed cause of contention.

References

1. Agrawal, K., Benoit, A., Dufossé, F., Robert, Y.: Mapping filtering streaming applications with communication costs. In: *SPAA*, pp. 19–28 (2009)
2. Agrawal, K., Benoit, A., Dufossé, F., Robert, Y.: Mapping filtering streaming applications. Algorithmica **62**(1–2), 258–308 (2012)
3. Boehm, M., Tatikonda, S., Reinwald, B., Sen, P., Tian, Yuanyuan, Burdick, D.R., Vaithyanathan, S.: Hybrid parallelization strategies for large-scale machine learning in systemml. Proc. VLDB Endow. **7**(7), 553–564 (2014)
4. Burge, J., Munagala, K., Srivastava, U.: Ordering pipelined query operators with precedence constraints. Technical Report 2005-40, Stanford InfoLab (2005)
5. Chaudhuri, S., Shim, Kyuseok: Optimization of queries with user-defined predicates. ACM Trans. Database Syst. **24**(2), 177–228 (1999)
6. Chirkin, A.M, Belloum, A.S.Z., Kovalchuk, S.V., Makkes, M.X.: Execution time estimation for workflow scheduling. WORKS '14, pp. 1–10 (2014)
7. Chirkin A.M., Belloum, A.S.Z., Kovalchuk, S.V., Makkes, M.X.: Execution time estimation for workflow scheduling. In: proceeding of the 9th Workshop on Workflows in Support of Large-Scale Science, pp. 1–10. IEEE Press (2014)
8. Deshpande, A., Hellerstein, L.: Parallel pipelined filter ordering with precedence constraints. ACM Transac. Algorithms **8**(4), 41:1–41:38 (2012)
9. DeWitt, D.J., Gray, J.: Parallel database systems: The future of high performance database systems. Commun. ACM, 35(6) (1992)

10. Florescu, D., Levy, A., Manolescu, I., Suciu, D.: Query optimization in the presence of limited access patterns. In: Proceedings of the 1999 ACM SIGMOD international conference on Management of data, SIGMOD '99, pp. 311–322. ACM (1999)
11. Halasipuram, R., Deshpande, P.M., Padmanabhan, S.: Determining essential statistics for cost based optimization of an ETL workflow. In EDBT, pp. 307–318 (2014)
12. Hellerstein, J.M.: Optimization techniques for queries with expensive methods. ACM Trans. Database Syst. **23**(2), 113–157 (1998)
13. Hueske, F., Peters, M., Sax, M., Rheinländer, A., Bergmann, R., Krettek, A., Tzoumas, K.: Opening the black boxes in data flow optimization. PVLDB **5**(11), 1256–1267 (2012)
14. Ibaraki, T., Kameda, T.: On the optimal nesting order for computing n-relational joins. ACM Trans. Database Syst. **9**(3), 482–502 (1984)
15. Kougka, G., Gounaris, A.: On optimizing workflows using query processing techniques. In: SSDBM, pp. 601–606 (2012)
16. Kougka, G., Gounaris, A.: Optimization of data-intensive flows: is it needed? is it solved? In: proceeding of the DOLAP, pp.95–98 (2014)
17. Kougka, G., Gounaris, A.: Cost optimization of data flows based on task re-ordering. T. Large-Scale Data- and Knowledge-Centered Systems **33**, pp. 113–145 (2017)
18. Kougka, G., Gounaris, A.: Optimal task ordering in chain data flows: exploring the practicality of non-scalable solutions. In Big Data Analytics and Knowledge Discovery-19th International Conference, DaWaK 2017, Lyon, August 28-31, 2017, Proceedings, pp. 19–32 (2017)
19. Kougka, G., Gounaris, A., Leser: Ulf Modeling data flow execution in a parallel environment. In: Big Data Analytics and Knowledge Discovery-19th International Conference, DaWaK 2017, pp. 183–196 (2017)
20. Kougka, G., Gounaris, A., Simitsis, A.: The many faces of data-centric workflow optimization: a survey. International Journal of Data Science and Analytics (2018)
21. Krishnamurthy, R., Boral, H., Zaniolo, C.: Optimization of nonrecursive queries. In: VLDB, pp. 128–137 (1986)
22. Kumar, N., Kumar, P.S.: An efficient heuristic for logical optimization of ETL workflows. In: BIRTE, pp.68–83 (2010)
23. Pietri, I., Juve, G., Deelman, E., Sakellariou, R.: A performance model to estimate execution time of scientific workflows on the cloud. WORKS '14, pp. 11–19. IEEE Press (2014)
24. Pietri, I., Juve, G., Deelman, E., Sakellariou, R.: A performance model to estimate execution time of scientific workflows on the cloud. In: Proceeding of the 9th Workshop on Workflows in Support of Large-Scale Science, pp. 11–19. IEEE Press (2014)
25. Poess, M., Rabl, T., Caufield, B.: TPC-DI: the first industry benchmark for data integration. PVLDB **7**(13), 1367–1378 (2014)
26. Rheinländer, A., Heise, A., Hueske, F., Leser, U., Naumann, Felix: SOFA: an extensible logical optimizer for UDF-heavy data flows. Inf. Syst. **52**, 96–125 (2015)
27. Rheinländer, A., Leser, U., Graefe, G.: Optimization of complex dataflows with user-defined functions. ACM Comput. Surv. **50**(3), 38:1–38:39 (2017)
28. Rheinländer, A., Heise, A., Hueske, F., Leser, U., Naumann, F.: Sofa: An extensible logical optimizer for udf-heavy data flows. Inform. Syst. **52**, 96–125 (2015)
29. Shi, J., Zou, J., Jiaheng, L., Cao, Z., Li, S., Wang, C.: Mrtuner: a toolkit to enable holistic optimization for mapreduce jobs. Proc. VLDB Endow. **7**(13), 1319–1330 (2014)
30. Simitsis, A., Vassiliadis, P., Sellis, T.K.: State-space optimization of ETL workflows. IEEE Trans. Knowl. Data Eng. **17**(10), 1404–1419 (2005)
31. Simitsis, A., Wilkinson, K., Dayal, U., Castellanos, M.: Optimizing ETL workflows for fault-tolerance. In: ICDE, pp. 385–396 (2010)
32. Singhal, R., Verma, A.: Predicting job completion time in heterogeneous mapreduce environments. In: IEEE IPDPSW, pp. 17–27 (2016)
33. Srivastava, U., Munagala, K., Widom, J., Motwani, R.: Query optimization over web services. In: Proceeding of the PVLDB, pp. 355–366 (2006)
34. Tziovara V., Vassiliadis, P., Simitsis, A.: Deciding the physical implementation of ETL workflows. In: DOLAP, pp. 49–56 (2007)
35. Verma, A., Cherkasova, L., Roy, H.: Campbell. Aria: Automatic resource inference and allocation for mapreduce environments. ICAC '11, pp. 235–244. ACM (2011)

36. Yerneni, R. , Li, C., Ullman, J.D., Garcia-Molina, Hector: Optimizing large join queries in mediation systems. In: ICDT, pp. 348–364 (1999)
37. Zhang, Z., Cherkasova, L., Loo, BT.: Performance modeling of mapreduce jobs in heterogeneous cloud environments. CLOUD '13, pp. 839–846 (2013)