Cost optimization of data flows based on task re-ordering

Georgia Kougka and Anastasios Gounaris

Department of Informatics
Aristotle University of Thessaloniki, Greece
{georkoug,gounaria}@csd.auth.gr

Abstract. Analyzing big data with the help of automated data flows attracts a lot of attention because of the growing need for end-to-end processing of this data. Modern data flows may consist of a high number of tasks and it is difficult for flow designers to define an efficient execution order of the tasks manually given that, typically, there is significant freedom in the valid positioning for some of the tasks. Several automated execution plan enumeration techniques have been proposed. These solutions can be broadly classified into three categories, each having significant limitations: (i) the optimizations are based on rewrite rules similar to those used in databases, such as filter and projection push-down, but these rules cover only the flow tasks that correspond to extended relational algebra operators. To cover arbitrary tasks, the solutions (ii) either rely on simple heuristics, or (iii) they exhaustively check all orderings, and thus cannot scale. We target the second category and we propose an efficient and polynomial cost-based task ordering solution for flows with arbitrary tasks seen as black boxes. We evaluated our proposals using both real runs and simulations, and the results show that we can achieve speed-ups of orders of magnitude, especially for flows with a high number of tasks even for relatively low flexibility in task positioning.

1 Introduction

Complex data analysis becomes more and more critical in order to extract high-quality information from raw data that is nowadays produced at an extreme scale. The ultimate goal is to derive actionable information in a timely manner. To this end, the usual practice is to employ fully automated data-centric flows (or simply called data flows) both for business intelligence [6] and scientific purposes [21]. The fact that data flows are typically data and/or computation intensive, combined with the volatile nature of the environment and the data, gives rise to the need for efficient optimization techniques tailored to data flows.

Data flows define the processing of large data volumes as a sequence of data manipulation tasks. An example of a real-world, analytic flow is one that processes free-form text data retrieved from Twitter (tweets) that comment on products in order to compose a dynamic report considering sales, advertisement campaigns and user feedback after performing a dozen of steps [26]. Example steps include the extraction of date information, quantifying the user sentiment through text analysis, filtering, grouping and expanding the information contained in the tweets through lookups in (static) data sources. Another example is to process newspaper articles, perform linguistic analysis, extract named entities and then establish relationships between companies and persons [24]. The tasks in a flow can either have a direct correspondence to operators of the extended relational algebra, such as filters, grouping, aggregates and joins, or encapsulate arbitrary data transformations, text analytics, machine learning algorithms and so on.

One of the most important steps in the data flow design is the specification of the execution order of the constituent tasks [22, 13, 26]. In practice, this can be the result of a manual procedure, which may result in non-optimal flow execution plans with regards to the sequence of tasks. Furthermore, even if a data flow execution plan is optimal for a specific input data set, it may prove significantly suboptimal for another data set with different statistical characteristics [11].

Automated plan enumeration solutions can be broadly classified into three categories. The first category is exemplified by the approaches followed by systems such as Pig [23] and JAQL, which utilize a rich set of rules to enhance an initial flow execution plan. These rules constitute a direct knowledge transfer from database query optimization, e.g., filter and projection push-down [10], but can cover only the tasks that have counterparts in the extended relational algebra. In general, data flow optimization is different from traditional query optimization in that the tasks do not necessarily belong to an algebra with clear semantics, and as such, are treated as black box, e.g., like user-defined functions (UDFs).

The second category consists of heuristics, e.g., [25, 34], and simple extensions of optimization techniques initially proposed for database queries with UDFs [12, 7]. A strong point of these plan enumeration solutions is that they can handle arbitrary tasks. Their weakest point is that they leave significant room for further improvements as proved in this work. The third category covers exhaustive solutions that are optimal for small flows but inapplicable to large flows because they do not scale with the size of the flow, e.g. [13].

In this work, we target the second category and we advance the state-of-the-art through the proposal of novel optimization algorithms that define the execution order of the tasks in a data flow in an efficient manner thus relieving the flow designers from the burden of selecting the task ordering on their own. The proposed solutions are applicable to large flows containing arbitrary data manipulation tasks and attain significantly better performance, i.e., they offer average speed-up of several factors, whereas in stand-alone cases, the speed-up can be up to two orders of magnitude. Our proposal refers to the logical level and is orthogonal to physical execution details; as such, it is applicable to both centralized and parallel execution environments.

The proposed optimization solutions were validated, as a proof of concept, in a real environment, namely Pentaho Data Integration (*PDI*), which is a widespread data flow tool [1]. Additionally, we performed thorough evaluations using synthetic data flows. The summary of our contributions is as follows:[1]

1. We introduce novel approximate low complexity algorithms that can be used for task reordering in data flows that have the form of a chain (Section 4). We show how we can further improve performance using optimizations that produce non-linear flow execution plans, where a task sends its output to several downstream tasks in parallel (Section 5).
2. We generalize the above results for flows with arbitrary number of sources and sinks thus covering any type of flow plans that are represented as directed acyclic graphs (DAGs) (Section 6).
3. We conduct thorough experiments using both real runs and simulations (Section 7). The evaluation results prove that the approaches introduced here significantly and consistently outperform the current state-of-the-art in all our experiments.

The remainder of this paper is structured as follows. A motivation case study is presented in Section 2. In Section 3, we formally introduce the main concepts, the problem, and its complexity.In Sections 4 and 5, our solutions for optimizing chain flows are analyzed. Optimization of more generic flows is discussed in Section 6. Section 7 presents the experimental analysis and finally, the discussion of the related work and the conclusions are presented in Sections 8 and 9, respectively.

2 Motivational Case Study

The purpose of this section is to demonstrate the inadequacy of existing approaches. We have implemented and executed a simple real-world chain (also referred to as *L-SISO*) flow in PDI, which consists of 13 tasks overall (configuration details are in Section 7). This flow retrieves tweets, extracts and analyzes tags referring to products, filters data and accesses static data sources in order to compose a dynamic report that associates sales with marketing campaigns (a full description can be obtained from [17]). Figure 1 presents the dependency constraints, which hold between tasks; the *DoF* in this case is 0.62, which is a representative value given that several real-world flows, as presented in [24], have *DoF* around 0.6. The dependency constraints and the *DoF* value are formally defined in the next section. Note that the four underlying static data sources are hidden behind the look-up tasks, and as such, the flow is treated as a chain.

[1] An extended abstract of some of the ideas in Section 4 has appeared in [16]. Also, in an earlier technical report [17], we have presented additional algorithmic descriptions and examples of most of the solutions in Sections 4 and 5 presented here. The material presented in this work and not appearing in [17] includes a thorough discussion of extensions to optimization techniques for database queries with UDF predicates, new techniques for multi-sink/source data flows, evaluation in a real system and fully revised synthetic experiments.

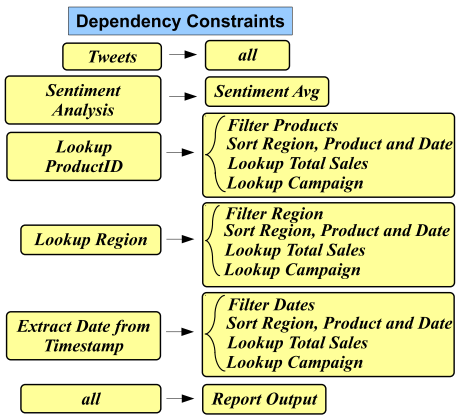


Fig. 1. The precedence constraints of the data flow in Figure 2.

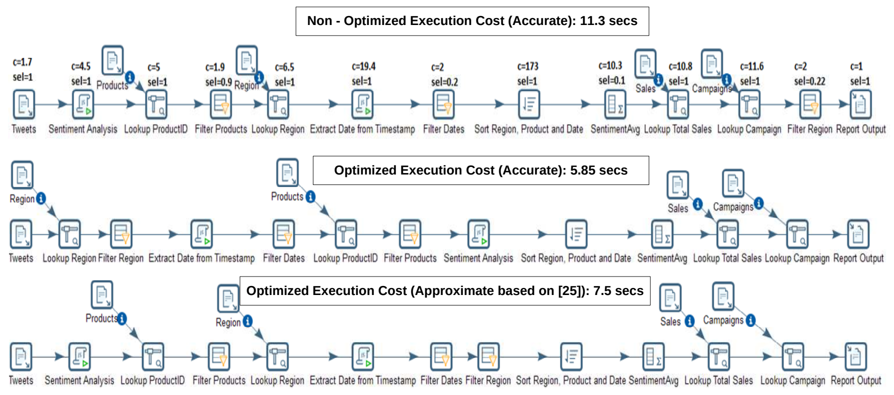


Fig. 2. Three alternative execution plans for our example data flow: the initial flow (top), the optimal one (middle) and the one produced by the technique in [25] (bottom)

In Figure 2(top), a straight-forward implementation is presented, along with the cost and selectivity values of each task. These values are extracted through profiling. In a setting similar to the one used in the evaluation (see Section 7), this flow is capable of processing 1 million tweets in 11.3 secs. Note that all constraints in Figure 1 are satisfied. In the middle of this figure, the optimal execution plan is presented, which can drop the execution cost approximately to the half. The optimal plan is generated through exhaustive enumeration of alternatives with the constrain that tasks on the left part of Figure 1 must precede those on the right part.

A question arises as to whether existing solutions of the first two categories in the introduction are capable of producing such a plan. Although, the plan is very simple, the answer is negative. First, due to the fact that the flow contains tasks not belonging to the extended relational algebra (e.g., extracting date with text processing, sentiment analysis, and so on), the solutions of the first category cannot perform any changes in the task ordering. The best performing scalable heuristics of the second category, according to the evidence of the evaluation in this work, has been proposed in [25]. This heuristic is based on greedy swaps of adjacent activities, if this plan transformation yields lower cost. The optimized plan is illustrated in Figure 2(bottom). In that case, the performance improvement is significant, but there is a clear gap between its plan and the optimal one.

The optimal execution plan for this scenario should move the filtering task *Filter Region*, which is initially the end, at the very beginning. Using methods as presented in [25], the region filter cannot move earlier unless the *campaign lookup* task is moved earlier as well due to the precedence constraints, an action that the greedy algorithm cannot cover. A less obvious optimization is to move the pair of date extraction and filtering tasks upstream although the former is expensive and not filtering. In the next section, we propose solutions that are capable of producing the optimal plan in this specific scenario and, further, they can scale to flows with hundreds of tasks.

Finally, exhaustive solutions cannot apply to flows with more than 20-25 tasks [17].

3 Preliminaries

In this paper, we deal with the problem of re-ordering the tasks of a data flow without violating possible precedence constraints between tasks, while the sum of the execution time for all tasks is minimized. The data flow is represented as a directed acyclic graph (DAG), where each task corresponds to a node in the graph and the edges between nodes represent intermediate data shipping among tasks.

3.1 Notation, Terminology and Assumptions

The main notation, terminology and assumptions are described as follows:

- Let $G = (T, E)$ be a DAG, where T denotes the nodes of the graph (that correspond to flow tasks) and E represents the edges (that correspond to the flow of data among the tasks). G corresponds to the execution plan of a data flow, defining one valid execution order of the tasks.
- $T = \{t_1, ..., t_n\}$ is a set of tasks[2] of size n. Each flow task is responsible for one or both of the following: (i) reading or retrieving or storing data, and (ii) manipulating data.

[2] In the remainder of the paper, we will use the terms tasks, services and activities interchangeably.

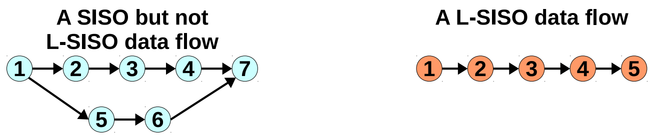


Fig. 3. Two examples of *SISO* data flows.

- Let $E = \{edge_1, ..., edge_m\}$ be a set of edges of size m. Each edge $edge_i, 1 \leq i \leq m$ equals to an ordered pair (t_j, t_k) denoting that task t_j sends data to task t_k, while m is less than or equal to $\frac{n(n-1)}{2}$; otherwise G cannot be acyclic.
- Let $PC = (T, D)$ be another DAG. D defines the precedence constraints (dependencies) that might exist between pairs of tasks in T. More formally, $D = \{d_1, ..., d_l\}$ 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. In other words, G should contain a path from t_j to t_k. Essentially, the PC graph defines constraints on the valid edges of the G graph. This also implies that if D contains (t_a, t_b) and (t_b, t_c), it must also contain (t_a, t_c).
  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. To avoid confusion, we use dotted arrows for PC edges.
- A *single-input single-output* (*SISO*) data flow is defined as a flow G that contains only one task with no incoming edges, termed as *source*, from another task and only one task with no outgoing edges, termed as *sink*. In a *SISO* flow, there is a dependency edge d from the source task to any other non-sink task, and from all non-source tasks to the sink task.
- A *L-SISO* flow is a specific form of a *SISO* data flow, where the tasks can form a chain of tasks. More specifically, each node of the G graph has exactly only one incoming and outgoing edge, while *source* and *sink* tasks have only one outgoing and incoming edge, respectively. In Figure 3, two examples of *SISO* flows are presented, where only the right one is a *L-SISO* flow.
  A *L-SISO* flow can be executed both as a linear and as a non-linear (parallel) flow, as Figure 4 shows. In linear flows, G has the form of a chain, and each non-source and non-sink task has exactly one incoming and one outgoing edge. In non-linear flows, the output of a single task can be fed to multiple tasks in parallel. In the non-linear examples in the figures, the tasks that receive more than one input have been augmented to include a merge function. In the rightmost example in Figure 4, the merge is a natural join on the outputs of tasks 2 and 3, whereas task 4 remains a unary task. More details on this are provided in Section 5.
- We define the *Degree-of-Freedom* quantity *DoF* for *L-SISO* flows as follows: $DoF = 1 - \frac{2l}{n(n-1)}$, where l is the size of set of dependencies as defined above and $DoF \in [0, 1]$. In the case that $DoF = 0$ then $l = \frac{n(n-1)}{2}$, which implies

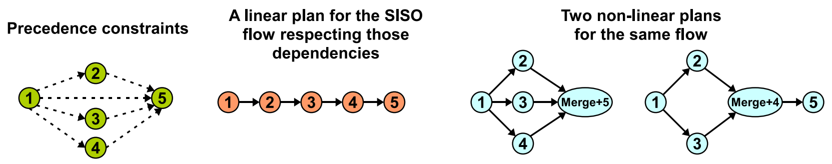

Precedence constraints
A linear plan for the SISO flow respecting those dependencies
Two non-linear plans for the same flow
Merge+5
Merge+4

Fig. 4. Examples of three execution plans of a *L-SISO* data flow.

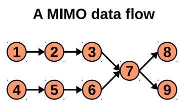


Fig. 5. A *MIMO* data flow.

that the flow is fully constrained and there is absolutely no flexibility in the ordering of flow tasks (i.e., there is only one valid flow execution plan). Additionally, when $DoF = 1$ denotes that flow tasks can be ordered in an arbitrary manner. In general, the higher the DoF value, the higher the need for efficient optimization. [3]

- In general, data flows are *multiple-input multiple-output MIMO* data flows, as shown in Figure 5. Each *MIMO* comprises a set of *L-SISOs*. In the example of the figure, there are multiple *L-SISOs*, e.g., $1 \rightarrow 2 \rightarrow 3 \rightarrow 7$, $4 \rightarrow 5 \rightarrow 6 \rightarrow 7$, $7 \rightarrow 8$, and $7 \rightarrow 9$. Also, generic *SISOs* consist of multiple *L-SISOs*, e.g., like the one in Figure 3(left).

In a data flow, we assume that each task receives some data items as an input and outputs some other data items as a result. Following the database terminology, each data item is referred to as a *tuple*. The task metadata that our optimization techniques require are:

- *Cost* (c_i): we use c_i, $1 \leq i \leq n$ as a metric of the time cost of each task per data item processed. This cost can also encapsulate the data transmission cost to the next tasks downstream.
- *Selectivity* (sel_i): it denotes the average number of returned data items per source tuple for the i-th service. For filtering operators, $sel_i < 1$, for data sources and operators that just manipulate the input $sel = 1$, whereas, for operators that may produce more output records for each input record, $sel_i > 1$. An example of a task with $sel_i < 1$ is a bank application that processes customers in order to report those with inactive accounts, while, an example of a task with $sel_i > 1$ is a bank procedure that outputs all the connected credit cards for a given account (assuming that each customer has more than one credit card on average).

[3] Note that when $DoF = 1$, which is rarely the case in data flows, standard database query optimization solutions become applicable, because then each task can be treated in the same way a filter is treated with the simple extension that selectivity can be higher than that.

- *Input* (inp_i): it denotes the size of the input of the i-th task t_i in number of tuples per input data tuple. It depends on the product of the selectivities of the preceding tasks in the execution plan G.[4] More formally, if T_i^{prec} is the set of all preceding tasks of t_i in G, $inp_i = \prod_{j=1}^{|T_i^{prec}|} sel_j$.
- *Output* (out_i): The size of the output of the i-th task per source tuple can be easily derived from the above quantities, as it is equal to $inp_i sel_i$.

Based on the above, each task is described as a triple $t_i =< c_i, sel_i, inp_i >$. Assuming that selectivities are independent, we can infer that inp_i is the only task characteristic that depends on the position of t_i in G; the cost and the selectivity of each task is independent of the exact G that may include t_i.

3.2 Problem Statement, Complexity, Optimality and Approach Overview

Problem Statement: Given an initial G graph, a set of tasks T with known cost and selectivity values, and a corresponding precedence constraint graph PC, we aim to find a valid G that minimizes the *sum cost metric (SCM)* per source tuple, defined as follows: *SCM(G)=* $inp_1c_1 + inp_2c_2 + ... + inp_nc_n$. The optimized plan is denoted as P. $\square$

SCM's rationale is to provide a good approximation of the resource consumption during the flow execution regardless of physical execution details. Furthermore, in the specific case where all tasks are executed sequentially, it provides a good approximation of the execution time. Also, it is a common metric in query optimization as well [10]. Typically the user is capable of providing an initial G and a set of precedence constraints, possibly with the help of techniques, such as those in [24]. From the initial set of these constraints, the full PC graph can be easily derived through the computation of its transitive closure.

Note that more sophisticated cost models can be considered, e.g., models that define the cost of a task as a function of the input bytes and consider in the selectivity the number of attributes added or removed. The issue of a data flow cost model is largely orthogonal to our optimization solutions and we leave the investigation of cost modelling approaches to future work.

Problem Complexity and Optimality: In [5], where query operators with precedence constraints that are equivalent to our tasks are considered, it is proved that finding the optimal ordering of tasks is an NP-hard problem when (i) each flow task is characterized by its cost per input record and selectivity; (ii) the cost of each task is a linear function of the number of records processed and that number of records depends on the product of the selectivities of all preceding tasks (assuming independence of selectivities for simplicity); and (iii) the optimization criterion is the minimization of the sum of the costs of all tasks. All the above conditions hold for our case, so our problem is intractable.

[4] Here, there is an implicit assumption that the selectivities are independent; if this is not the case, the product will be an arbitrarily erroneous approximation of the actual selectivity of the subplan before each task.

Moreover, in [5] it is discussed that *"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 such, in this work, we do not make any formal claim as to how close the optimized plans derived by our solutions are to the optimal plans in the generic case. Note that if we modify the optimization criterion, e.g., to optimize the bottleneck cost metric or the critical path renders the problem tractable [28, 2].

Our approach in a nutshell: To cope with the problem complexity, we adopt a divide-and-conquer technique. From the initial G, which is either a *MIMO* or a generic *SISO*, we extract *L-SISO* segments. Then, we optimize *L-SISOs* in polynomial time independently.

4 Optimization of *L-SISO* flows

To tackle the limitations of existing solutions, exemplified in Section 2, we aim to develop approximate solutions that will be scalable for medium and large data flows while improving the performance. In the first subsection, we present solutions that are essentially extensions of existing UDF query optimization proposals so that they become applicable to our problem. Then, we present our main novelty with regards to approximate optimization of linear data flows.

The main rationale for optimizing linear plans for *L-SISO* data flows is described, as follows. Firstly, we employ the join ordering algorithm proposed in [14, 19] as a core building block, which will be referred to as *KBZ*. The reason we choose *KBZ* is to avoid re-inventing the wheel and benefit from the existing proposals in query optimization to the largest possible extent. *KBZ* leverages the rank value of each task defined as $\frac{1-sel_i}{c_i}$, and also considers the dependencies among tasks. When there are no dependencies, it is known from database optimization research that ordering the tasks by their rank value yields the optimal execution plan [14, 19]. The main approach of *KBZ* to handling the cases where such ordering is not possible because a task with a lower rank should precede a task with a higher rank is to merge these tasks. For example, assume that we merge t_i and t_j. Then, the cost of the merged task becomes $c_{ij} = c_i + sel_i c_j$ and the new selectivity is $sel_{ij} = sel_i sel_j$. The rank value of the merged task equals to $\frac{1-sel_{ij}}{c_{ij}}$. By successively merging tasks if needed, we ensure that all the remaining tasks are ordered by rank. This allows re-orderings of sets of tasks rather than individual tasks, and thus is capable of producing the optimal execution plan for the example shown in Figure 2. The time complexity of *KBZ* algorithm is $O(n^2)$.

Our solutions can be described at a high-level as shown in Algorithm 1. The main challenge is to devise an efficient preprocessing technique, so that *KBZ* becomes applicable, since *KBZ* does not account for arbitrary dependencies between tasks. This step applies to the *PC* graph. A trivial step in that preprocessing phase is to remove all edges that can be inferred through the transitive closure. In addition, we need to post-process the result of the *KBZ* algorithm in order either to guarantee validity or to further improve the intermediate results.

```
Algorithm 1 Rank ordering based high-level algorithm
Require: A set of n tasks, T={$t_1$, ..., $t_n$} and the PC graph
Ensure: A directed acyclic graph P representing the optimal plan
 1: Pre-processing phase: modify PC
 2: Apply KBZ algorithm, so that a G is produced.
 3: Post-processing phase: enhance G
 4: Set P← final G
```

There are many options regarding how these two phases can be performed and in this section, we present three concrete suggestions, which constitute the novelty of this section.

Also note that the input does not require an initial G graph. This is because, for each *L-SISO*, respecting the precedence constraints is a necessary and sufficient validity condition for producing valid flows.[5]

4.1 Extending Solutions for Queries with UDFs

Approximate optimization solutions have been proposed for queries containing UDFs in [7],[12], which leverage the task rank values as well. These techniques cannot be applied to the data flow optimization problem in their original form because the dependency constraints that they consider refer to pairs of a join and a UDF, rather than between UDFs. However, it is straightforward to apply extensions so that constraints between UDFs are taken into account; then we can treat flow tasks as UDFs.

The rationale of the technique in [7] is to order the tasks in descending order of their rank values provided that the dependencies are respected. Our proposed extension adopts this rationale as well, through applying a greedy algorithm. Specifically, the greedy extension starts from a plan containing only the source task and in each step, adds the task with the maximum rank, as defined above, without violating the precedence constraints. The extension of the technique in [7] is essentially the same as the greedy optimization solution, called *GreedyI*, which is thoroughly presented and evaluated in [17], where it is shown that *GreedyI* is outperformed by *Swap* [25].

Similarly, we extend the predicate migration algorithm [12], denoted as *PM-based*, in order to optimize data flows considering dependency constraints between tasks. The proposal in [12] states that tasks should be ordered by their rank values, and if a task with a lower rank is prerequisite of a task with a higher rank value, the former should be placed just before the latter. In order to apply this rationale to data flows, initially, we sort the tasks based on their rank values without taking into account the precedence constraints. In the post-processing

[5] Respecting the precedence constraints is not sufficient for generic flows. For example, for the left flow in Figure 3, the precedence constraints as defined in this work cannot capture the requirement that tasks 2,3 and 4 should be placed in different branches than tasks 5 and 6.

phase, we detect possible constraint violations and resolve them by transposing the prerequisite tasks just before the task that they must precede. If the tasks that violate the existing constraints are more than one, they are transposed in the exact order they are initially positioned. This technique is proved less efficient than the rank ordering heuristics that we propose below, as shown in the evaluation in Section 7.

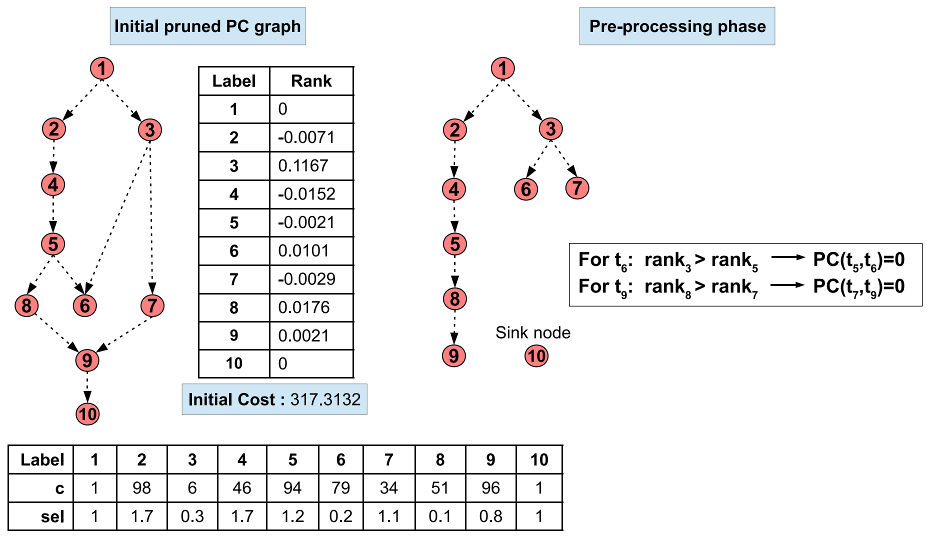
4.2 Our first rank ordering-based algorithm RO-I

Our main proposals do not merely extend UDF-query optimization but build upon the *KBZ* algorithm. In its original form, *KBZ* algorithm considers only a specific form of precedence constraints, namely those representable as a rooted tree. The fact that *KBZ* algorithm allows only tree-shaped precedence constraint graphs implies that there should be no task with more than one independent prerequisite activity, and in such data flow scenarios, the *DoF* is very high and increases more with the number of tasks (e.g., more than *DoF*=0.9 for a 100-node flow). Both of these cases do not occur frequently in practice.

In our first proposal, called *RO-I* (standing for *Rank Ordering-based I*), the pre-processing phase ensures the transformation of the precedence constraint graph into a tree-shaped one. This is done by maintaining the incoming edge with the highest rank and removing the other edges, if a task has more than one incoming edge. This process allows *KBZ* to run but may produce invalid flow orderings, due to the removal of dependencies during pre-processing. To fix that, we employ a post-processing phase where any resulting precedence constraint violations are resolved by moving tasks upstream if needed as prerequisites for other tasks placed earlier.

The worst case complexity of the pre-processing phase is $O(n^2)$ because we remove up to $n-1$ incoming edges from each task and we repeat this for $n-1$ tasks of the flow. Additionally, in the post-processing step, we check, for each of the n tasks, if any of the preceding tasks violates the precedence constraints. There can be up to $n-1$ preceding tasks in a flow ordering. So, in the worst case, the complexity is $O(n^2)$.

An illustrative example of applying *RO-I* is depicted in Figure 6. In that figure, we present the initial metadata and the pre-processing phase, in order to transform the precedence constraint graph into tree-shaped graph. The initial precedence constraint graph is pruned in the sense that all edges implied by the transitive closure are removed. The graph on the right side of the figure shows the final result after removing the edges (t_5, t_6), (t_7, t_9) (as shown in the figure, it is convenient to temporarily leave the sink task out, and after optimization, to attach it to the last task). Then, we apply the *KBZ* algorithm and finally, we apply the validity post-process phase of *RO-I* to ensure that the optimized execution flow plan does not violate the dependency constraints. In the transformed graph, KBZ has no knowledge for example, that t_6 should not be placed before t_5, and indeed it initially places t_6 in a way that violated the constraints. Figure 7 shows the complete way in which the output of *KBZ* on the top is transformed to a valid plan.


Initial pruned PC graph
Pre-processing phase
Initial Cost : 317.3132
For t_6: rank$_3$ > rank$_5$ ⟶ PC(t_5,t_6)=0
For t_9: rank$_8$ > rank$_7$ ⟶ PC(t_7,t_9)=0
Sink node

Label	Rank
1	0
2	-0.0071
3	0.1167
4	-0.0152
5	-0.0021
6	0.0101
7	-0.0029
8	0.0176
9	0.0021
10	0

Label	1	2	3	4	5	6	7	8	9	10
c	1	98	6	46	94	79	34	51	96	1
sel	1	1.7	0.3	1.7	1.2	0.2	1.1	0.1	0.8	1

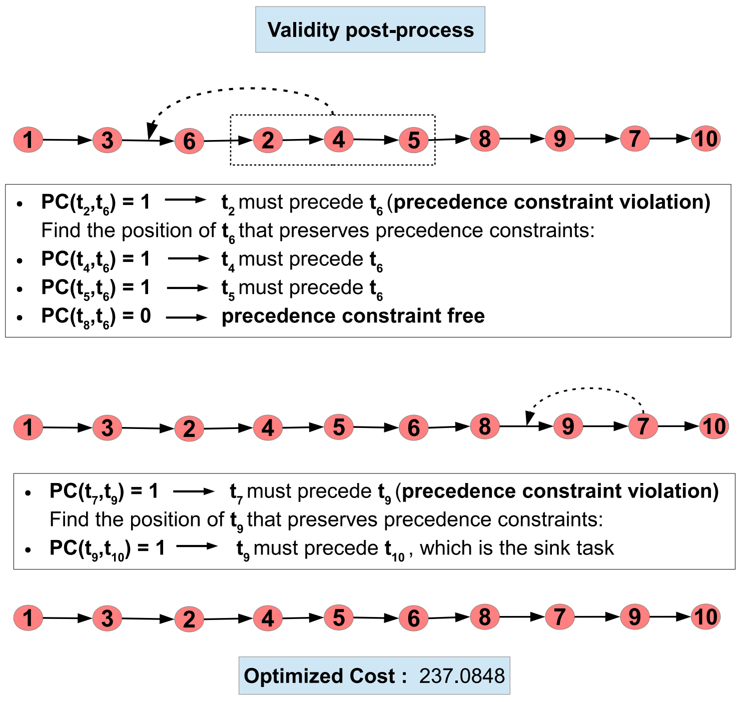
Fig. 6. An example of *RO-I* pre-processing phase.

4.3 RO-II

The *RO-II* algorithm follows a different approach in order to render *KBZ* applicable, that is to transform the *PC* graph to a tree. In the pre-processing phase, this approximate algorithm first detects paths in the precedence constraint graph that share an intermediate source and sink. Then it merges them to a single path based on their rank values. When there are multiple such paths, we start merging from the most upstream ones and when there are nested paths, we start merging from the innermost ones. In that way, all precedence constraints are preserved at the expense of implicitly examining fewer re-orderings. Figure 8 illustrates in detail the steps of the application of *RO-II* in the flow of Figure 6. The steps 1-3 describe the pre-processing phase of *RO-II*, where we merge two sub-segments into a linear sub-flow, because they create cycles by sharing the same intermediate source and sink. In that example, after the merging procedure we enforce more precedence constraints than the original ones, so that the task t_3 must precede not only task t_6 and t_7 but also tasks t_2, t_4, t_5 and t_8. In other words, the merging process imposes more restrictions on the possible re-orderings. As such, these local optimizations may still deviate from a globally optimal solution significantly in the average case. *RO-II* does not require any post-processing because its result is always valid.

In the case of *RO-II*, the time complexity remains $O(n^2)$ because, for each merge process, we consider at most $O(n)$ flow tasks and we repeat this for all the possible merge processes that can be up to n.

4.4 RO-III: an enhancement to RO-II



Validity post-process
1 3 6 2 4 5 8 9 7 10
PC(t2,t6) = 1 → t2 must precede t6 (precedence constraint violation)
Find the position of t6 that preserves precedence constraints:
PC(t4,t6) = 1 → t4 must precede t6
PC(t5,t6) = 1 → t5 must precede t6
PC(t8,t6) = 0 → precedence constraint free
1 3 2 4 5 6 8 9 7 10
PC(t7,t9) = 1 → t7 must precede t9 (precedence constraint violation)
Find the position of t9 that preserves precedence constraints:
PC(t9,t10) = 1 → t9 must precede t10 , which is the sink task
1 3 2 4 5 6 8 7 9 10
Optimized Cost : 237.0848

Fig. 7. The post-processing phase of *RO-I* for the example in Figure 6.

Algorithm 2 RO-III post-processing phase

```
Require: A set of n tasks, T={$t_1$, ..., $t_n$}
    A directed acyclic graph with precedence constraints
    Optimized plan G as a directed acyclic graph returned by RO-II
Ensure: A directed acyclic graph P representing the optimal plan
 1: repeat
 2:   {$k$ is the maximum subplan size considered}
 3:   for i=1:k do
 4:     for s=1:n-i do
 5:       for t=s+i:n do
 6:         consider moving subplan of size $i$ starting from the $s^{th}$ task after the $t^{th}$
            task in G
 7:       end for
 8:     end for
 9:   end for
10: until no changes applied
11: P ← G
```

After the evaluation of the proposed *RO-I* and *RO-II* algorithms, we isolated data flows where optimization yielded not near-optimal plans. A typical problem

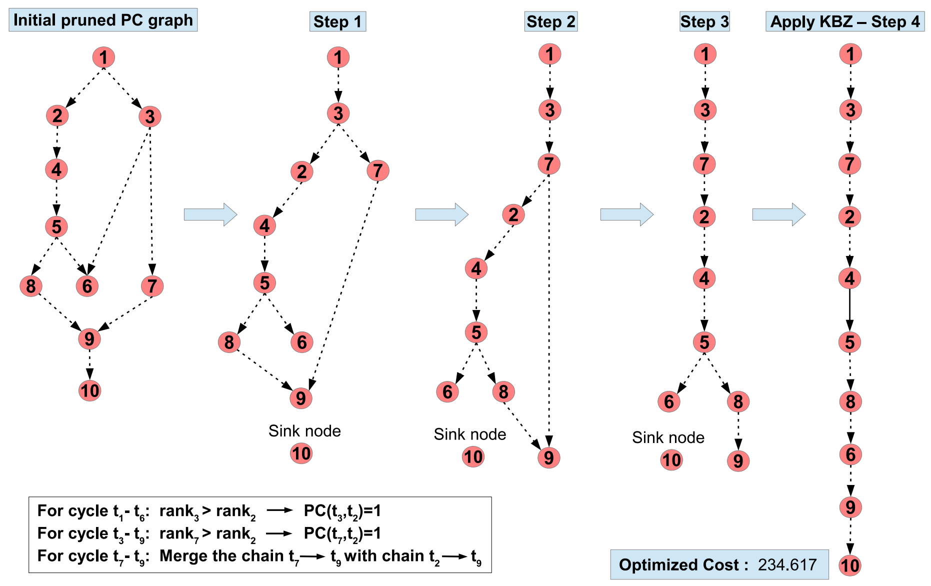
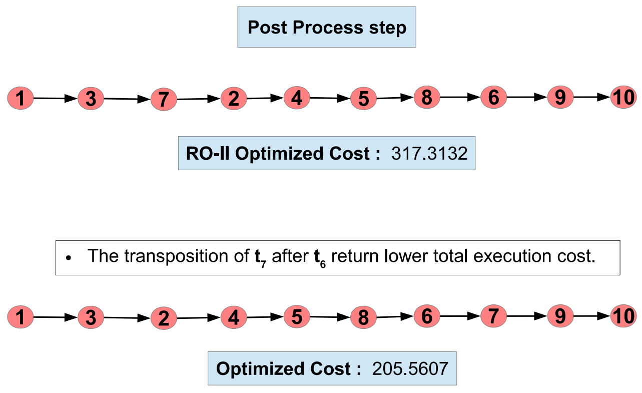

Initial pruned PC graph
Step 1
Step 2
Step 3
Apply KBZ – Step 4
Sink node
Sink node
Sink node
For cycle t_1- t_6: $rank_3 > rank_2 \longrightarrow PC(t_3,t_2)=1$
For cycle t_3- t_9: $rank_7 > rank_2 \longrightarrow PC(t_7,t_2)=1$
For cycle t_7- t_9: Merge the chain $t_7 \longrightarrow t_9$ with chain $t_2 \longrightarrow t_9$
Optimized Cost : 234.617

Fig. 8. An application example of *RO-II* with the metadata of Figure 6.

with *RO-II* is that it cannot move a filtering task to an earlier stage of the flow, even if this is not constrained by operator precedences, due to the additional restrictions that are implicitly incorporated as explained earlier. To address this problem, we propose RO-III, which tackles the limitations of *RO-II* with the help of a post-processing phase that we introduce (see Algorithm 2). We first apply the *RO-II* algorithm in order to produce an intermediate execution plan, and then we examine several transpositions. More specifically, we check all the possible re-orderings of each sub-flow of size from 1 to k tasks in the plan. The checks are applied from the left to the right. In this way, we address the problem of a task being "trapped" in a suboptimal place upstream in the flow execution due to the additional implicit constraints introduced by *RO-II*. This process is described by the three nested for loops in Algorithm 2 and is repeated until there are no changes in the flow plan. The reason we repeat it is because each applied transposition may enable further valid transpositions that were not initially possible.

The post-processing phase of the *RO-III* algorithm has $O(kn^2)$ complexity, which is derived by the maximum number each of the three inner loops can execute. The *repeat* process in theory can execute up to n times, but according to our observations during experiments, we see that even for large flows, there is no change after 3 times. In all experiments, we set k to 5.

In Figure 9, the result of the post-processing phase of algorithm *RO-III* is described. In this phase the optimized flow plan occurred by moving t_7 to a later stage.



Post Process step
1 3 7 2 4 5 8 6 9 10
RO-II Optimized Cost : 317.3132
• The transposition of t7 after t6 return lower total execution cost.
1 3 2 4 5 8 6 7 9 10
Optimized Cost : 205.5607

Fig. 9. The post-process phase of the *RO-III* algorithm taking as input the generated optimized execution plan of *RO-II*, as depicted in 8.

4.5 Discussion

The above algorithms explore a different and overlapping search space. There is no clear winner between *RO-I* and *RO-II*, but, in general, both of them outperform the solutions in Section 4.1, as will also be discussed in Section 7. By design, *RO-III* explores the largest search space and always outperforms *RO-II* (since *RO-III* is an extension to *RO-II*), as *RO-II* may fail to reorder a filtering task to an earlier stage of the flow. But there might be cases where *RO-III* is inferior to *RO-I* or even *Swap*, as a consequent of the fact that different plans are indirectly and directly evaluated by each technique. However, as elaborated in the evaluation section, such cases are relatively rare.

5 Producing Non-linear Plans for *L-SISO* flows

This section focuses on the advantages of non-linear (or parallel) execution plans for *L-SISO* flows. As we have explained in Section 3, in a parallel flow, each single task can have multiple outgoing edges, which implies that the output of such a task is fed, as input, to multiple tasks. In the right part of Figure 4, we observe that a single task may have not only multiple outgoing edges, but also multiple ingoing edges. In this case, a single task receive as input data the output of multiple tasks, and merges them back into a single input. This is in line with the *AND-Join* workflow pattern as presented in [31], where the outgoing edge of multiple tasks that are executed in parallel converge into a single task.

In software tools, such as PDI, the merge process can be implemented by incorporating a common *sort merge self-join* on the record ids. A similar approach is followed also in flows consisting of calls to Web Services [28]. As such, merging multiple input streams incurs an extra execution cost. To assess this cost, we evaluated parallel data flows that were executed with the PDI tool. The conclusion was that the merge task cost has a small effect on the total flow execution

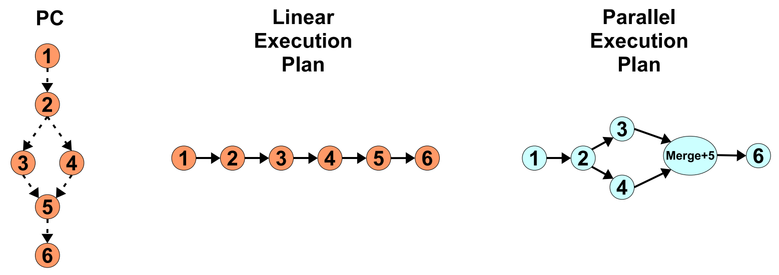
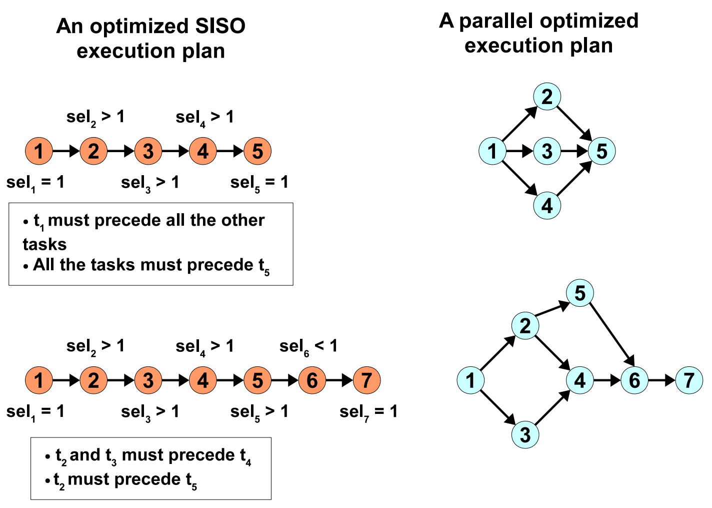


Fig. 10. Example of linear and non-linear execution plans for the same flow.

cost due to the fact that the inputs are typically already ordered by their IDs; in other words, the merge task is similar to an additional lightweight activity. Additionally, the size of the input (inp_i) of a task t_i, which receives more than one incoming edge is defined similarly to the tasks with only one incoming edge, i.e., by computing the product of the selectivity values of the preceding tasks as we have described in Section 3.

We now analyze the benefits of parallel flow execution by means of a theoretical example that considers two subsequent tasks t_3 and t_4 illustrated in Figure 10, which are not in a precedence relation and an extra cost of the merge process that will be denoted as mc. In this figure, we show two alternative plans, a linear one (in the middle) and a parallel one (on the right). The SCM values of the two alternatives vary only with respect to activities t_4 and t_5. We distinguish between the following four cases (using a superscript to differentiate the inputs in the two cases):

- Case I: $sel_3 \leq 1$ and $sel_4 \leq 1$. The linear execution cost is lower than the parallel execution cost, because (i) $inp_4^{linear} c_4 < inp_4^{parallel} c_4$ as $inp_4^{linear} = sel_3 inp_4^{parallel}$ and $sel_3 < 1$, and (ii) $inp_5^{linear} c_5 < inp_5^{parallel}(c_5 + mc)$ due to the extra merge cost of the parallel version and given that $inp_5^{linear} = inp_5^{parallel}$. So, in that case, parallelism is not beneficial.
- Case II: $sel_3 \leq 1$ and $sel_4 > 1$. Similar with the Case I, the linear execution of the flow is more beneficial than the parallel; note that the selectivity value sel_4 does not affect the previous statements.
- Case III: $sel_3 > 1$ and $sel_4 > 1$. If $mc = 0$, the parallel execution results in better performance than the linear execution. In that case $inp_5^{linear} c_5 = inp_5^{parallel}(c_5 + mc)$. Because of the fact that $sel_3 > 1$, we deduce that $inp_4^{linear} c_4 > inp_4^{parallel} c_4$. In the generic case where $mc > 0$, we need to compute the estimated costs in order to verify which option is more beneficial, but we expect that, for small mc values, the parallel execution to outperform.
- Case IV: $sel_3 > 1$ and $sel_4 \leq 1$. Following the rationale of the previous case, there is no clear winner between the two executions shown in Figure 10. However, an optimized linear plan will place t_4 before t_3 thus corresponding to Case II, where the (new) linear plan is better than the parallel one.



An optimized SISO execution plan
A parallel optimized execution plan
sel$_2$ > 1
sel$_4$ > 1
sel$_1$ = 1
sel$_3$ > 1
sel$_5$ = 1
• t$_1$ must precede all the other tasks
• All the tasks must precede t$_5$
sel$_2$ > 1
sel$_4$ > 1
sel$_6$ < 1
sel$_1$ = 1
sel$_3$ > 1
sel$_5$ > 1
sel$_7$ = 1
• t$_2$ and t$_3$ must precede t$_4$
• t$_2$ must precede t$_5$

Fig. 11. Example of executing *L-SISO* flows in parallel.

Algorithm 3 Post-process step for parallel *L-SISO* flows

```
Require: An optimized linear plan P={$t_1 \rightarrow ... \rightarrow t_n$}
         A directed acyclic graph with precedence constraints
Ensure: A directed acyclic graph P representing the optimal parallel plan
 1: i=1
 2: while $i < n$ do
 3:   j=i+1
 4:   while $sel_{P(j)} > 1$ do
 5:     Delete the edge between the tasks $t_{P(j-1)} \rightarrow t_{P(j)}$ from $P$
 6:     if $t_{P(j)}$ is not predecessor in precedence constraint graph for no task in
        $t_{i+1} \dots t_{j-1}$ then
 7:       Connect the edge between the tasks (i) $t_{P(i)}$ and (ii) $t_{P(j)}$, i.e., create the
          edge $t_{P(i)} \rightarrow t_{P(j)}$ in $P$
 8:     else
 9:       Connect in $P$ the edge between (i) all the preceding tasks in precedence
          constraint graph with no outgoing edges in $P$ and (ii) $t_{P(j)}$
10:     end if
11:     $j = j + 1$
12:   end while
13:   Connect in $P$ the edge between (i) all the tasks $t_{P(i+1)} \dots t_{P(j-1)}$ with no out-
      going edges in $P$ and (ii) $t_{P(j)}$
14:   i=j
15: end while
```

In order to exploit the advantages of the optimization opportunities of Case III, we introduce a post-process phase in the solutions of Section 4 (see Algorithm 3). After the generation of an optimized linear execution plan, we apply a post-process step that restructures the flow in a way that subsequent tasks having selectivity greater than 1 are executed in parallel if this does not incur violations of the precedence constraints. This post-process step can be applied to any optimization algorithm that produces a linear ordering.

Two examples are presented in Figure 11, where, in the upper flow scenario, we choose to parallelize t_2, t_3 and t_4, while in the flow case that is depicted in the bottom of the figure, we execute parallel only t_2 and t_3 and not t_4, because of the precedence constraints. Then, t_5 is appended after t_2 because of the constraints and is executed in parallel with t_4. As t_6 has selectivity value < 1, it is not executed in parallel with any other task.

The complexity is $O(n^2)$. The parallelization of each task is examined at most once, and for each such case, the preceding tasks need to be checked, the number of which cannot exceed n.

As a final note, in the previous discussion, we have silently assumed that sending the output to more than one task downstream does not incur an extra cost. This is common to centralized and share-memory parallel systems where the results of a task are kept in memory and are accessible to any subsequent task at no extra cost. In distributed settings where tasks are at distinct places, having multiple outgoing edges incurs extra cost, which however can be treated exactly as *mc*.

6 Optimizing *MIMO* flows: the complete approach

So far we have discussed the case of chains with single-source and a single-sink task, but arbitrary *multiple-input multiple-output (MIMO)* flows can benefit from the solutions presented in the previous sections. The generic types of *MIMO* flows are described in [32], two of which are shown in Figure 12. A main difference between *L-SISO* and *MIMO* flows is that apart from re-ordering tasks, additional optimization operations can apply. As explained in [25], the *factorize* and *distribute* operations can move an activity appearing in both input subflows of a binary activity to its output and the other way around, respectively.[6] This allows for example a filtering operation initially placed after a merge task to be pushed down to the merge inputs (provided that the filtering condition refers to both inputs), which is known to yield better performance.

Figure 12 displays exemplary MIMO data flows of types butterfly (top) and fork (bottom). In these cases, the optimization of *L-SISO* data flows can play an important role in optimizing *MIMO* flows as *MIMO* flows consist of a set of *L-SISO* flows. Algorithm 4 describes a divide-and-conquer proposal for optimizing *MIMO* flows, which is based on the extraction of these linear segments of the flow and apply optimization algorithms only on the *L-SISO* sub-flows. Remember that in the generic case, we start from an initial valid plan rather than from the *PC* graph.

[6] [25] additionally considers the case that an activity can be further split in several sub-activities, which is not considered here.

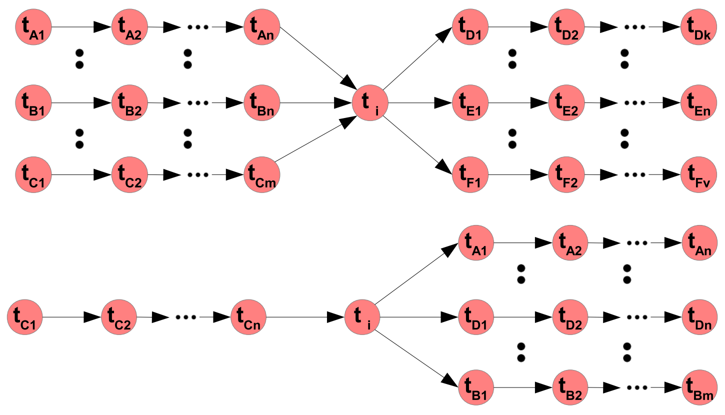


Fig. 12. Example *MIMO* data flows of type butterfly (top) and fork (bottom).

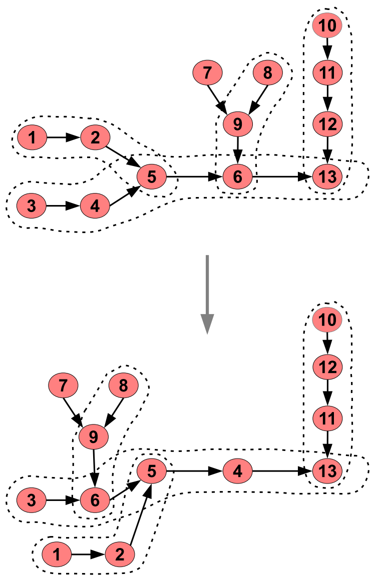
Algorithm 4 Optimization of *MIMO* flows

```
1: repeat
2:    Extract L-SISO segments
3:    for all L-SISO segments do
4:       Optimize L-SISO segments
5:    end for
6:    Apply factorize/distribute optimization thus modifying the L-SISO segments
7: until no changes
```

The extraction of the linear segments of a MIMO flow in line 2 of the algorithm can be performed in a simple manner with linear complexity. Traversing a path from a source, we stop when a task with multiple incoming edges is encountered (e.g., t_i in Figure 12). The latter task plays the role of the sink for that segment. It also plays the role of the source for each segment starting from it. We iteratively repeat this process until all tasks are visited.

In lines 3-5, we exploit the optimization solutions that we proposed for *L-SISO* flows and use them as the main building block to optimize *MIMO* flows. Then, we check whether we can apply the factorize/distribute operations, which modify the linear segments. This process is repeated until it converges. In this work, we focus solely on task re-ordering (which corresponds to optimize the linear segments individually) and the investigation of further techniques that combine task re-orderings with additional operations is left for future work.

The complexity of Algorithm 4 is quadratic in the number of tasks of the largest *L-SISO* segment and also quadratic in the number of *L-SISO* segments, given that the *factorize* and *distribute* operations can occur an amount of times that is proportional to the number of segments.



- **L-SISO with maximum path length (branch):**
  L-SISO segment : 1 – 2 – 5 – 6 – 13 (1)
  L-SISO segment : 3 – 4 – 5 – 6 – 13 (2)
- **L-SISO with minimum sum of ranks:**
  rank(1) > rank(2)
- **Optimize L-SISO (2):**
  L-SISO segment : 3 – 4 – 5 – 6 – 13
  Optimized L-SISO: 3 – 6 – 5 – 4 – 13
- **L-SISO with maximum path length:**
  L-SISO segment : 10 – 11 – 12 – 13 (3)
  Optimized L-SISO: 10 – 12 – 11 – 13
- **L-SISO with maximum path length (branch):**
  L-SISO segment : 7 – 9 – 6 (4)
  L-SISO segment : 8 – 9 – 6 (5)
- **L-SISO with minimum sum of ranks:**
  rank(4) > rank(5)
- **Optimize L-SISO (5):**
  L-SISO segment : 8 – 9 – 6
  Optimized L-SISO: 8 – 9 – 6
- **L-SISO with maximum path length:**
  L-SISO segment : 1 – 2 – 5 (6)
  Optimized L-SISO: 1 – 2 – 5

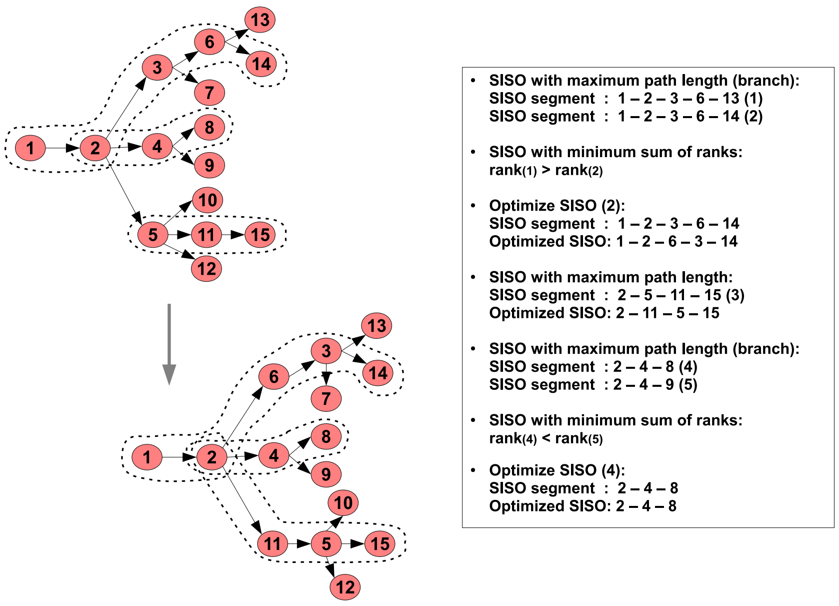
Fig. 13. An example of *MISO* data flow optimization.

6.1 The special case of SIMO and MISO flows

The *single-input multiple-output (SIMO)* and *multiple-input single-output (MISO)* are two special structures of *MIMO* data flows. In the following, we introduce two techniques for optimizing such data flow cases, when no *factorize* and *distribute* operations are required, e.g., there are no cases where a filter after a binary join should be moved to both input branches upstream. To perform the optimizations below, we relax the definition of *L-SISOs* to include any chain of operators so that larger segments in the flow are optimized using the *RO-III* algorithm (Section 4).

In the case of the *MISO* data flows, like the one in Figure 13, the first step is to find the *L-SISO* flow segment with the maximum length of path defined from the source to the sink task. In the case that we find two *L-SISO* segments with the same path length, we choose to optimize the sub-flow with the minimum sum of the rank values of the tasks that it consist of. This implies that we choose the *L-SISO* sub-flow with the most expensive tasks with high selectivity values.

A detailed example of our technique is described in Figure 13. In the figure, the *L-SISO* segments are in dotted boundaries, omitting the trivial ones that consist of a source and a sink task only. In this example, the *L-SISO* segment $t_1 \rightarrow t_2 \rightarrow t_5 \rightarrow t_6 \rightarrow t_{13}$ ***(1)*** and $t_3 \rightarrow t_4 \rightarrow t_5 \rightarrow t_6 \rightarrow t_{13}$ ***(2)*** have equal length of path from their source tasks to their common sink task. So, we estimate the sum of the rank values of sub-flow *(1)* and *(2)* showing that the minimum sum of ranks corresponds to the *L-SISO* segment $t_3 \rightarrow t_4 \rightarrow t_5 \rightarrow t_6 \rightarrow t_{13}$ ***(2)***.



- SISO with maximum path length (branch):
  SISO segment : 1 – 2 – 3 – 6 – 13 (1)
  SISO segment : 1 – 2 – 3 – 6 – 14 (2)
- SISO with minimum sum of ranks:
  rank(1) > rank(2)
- Optimize SISO (2):
  SISO segment : 1 – 2 – 3 – 6 – 14
  Optimized SISO: 1 – 2 – 6 – 3 – 14
- SISO with maximum path length:
  SISO segment : 2 – 5 – 11 – 15 (3)
  Optimized SISO: 2 – 11 – 5 – 15
- SISO with maximum path length (branch):
  SISO segment : 2 – 4 – 8 (4)
  SISO segment : 2 – 4 – 9 (5)
- SISO with minimum sum of ranks:
  rank(4) < rank(5)
- Optimize SISO (4):
  SISO segment : 2 – 4 – 8
  Optimized SISO: 2 – 4 – 8

Fig. 14. An example of *SIMO* optimization.

Then, we apply the *RO-III* algorithm to optimize this sub-flow by re-ordering its tasks, as shown in the bottom part of the figure.

After this optimization, the incoming edges of each of the tasks in the optimized subflow become immutable even when they come from external tasks, for example the edges $t_2 \to t_5$ and $t_9 \to t_6$ remain fixed. The corresponding tasks (t_5 and t_6) play the role of temporary sinks for the non-optimized branches. The next step is to isolate all the *L-SISO* segments that their tasks are not part of the already optimized *L-SISO* segment *(2)*, except for their temporary sinks that are tasks of the segment *(2)* in the example, e.g. $t_1 \to t_2 \to t_5$. For each of these, we follow the same procedure described earlier. Specifically, we find the maximum path length from each source task to its corresponding temporary sink. So, in the example, we optimize the *L-SISO* $t_{10} \to t_{11} \to t_{12} \to t_{13}$ *(3)*. Then, for equivalent *L-SISO* segments, such as $t_7 \to t_9 \to t_6$ and $t_8 \to t_9 \to t_6$, we follow the same approach until we finish with all branches.

For this technique to be correct, the precedence constraints need to be defined carefully. For instance, in the example of the figure, t_4 should be allowed to move downstream regarding the input of not only t_3 but t_1, t_7 and t_8, too.

Another special structure of *MIMO* flows is the *SIMO* one, where their optimization is based on the same rationale of the technique that we have just described. Figure 14 shows an example of the proposed technique. In this case, the main difference is that we initially consider one *L-SISO* segment for each sink. For example, the *L-SISO* segment $t_1 \to t_2 \to t_3 \to t_6 \to t_{13}$ and

$t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_6 \rightarrow t_{14}$ or $t_1 \rightarrow t_2 \rightarrow t_4 \rightarrow t_8$ and $t_1 \rightarrow t_2 \rightarrow t_4 \rightarrow t_9$, and so on. Then, we follow the same optimization procedure as for *MISO* data flows.

The main difference between these two techniques is that each segment reordering, like the reordering of the $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_6 \rightarrow t_{14}$ segment, may force the temporary source to change in order to avoid constraint violations for the remainder segments; the temporary source of the non optimized segment s belonging to the optimized segment s', is the most downstream vertex of s', which is connected to any vertex of s with a precedence constraint. For example, in Figure 14, t_{13} is connected to t_3 in the optimized plan, assuming that, initially, there is a precedence constraint between both (i) t_3 and t_{13} and (ii) t_6 and t_{13}. Further in this example, we assume that moving t_{11} before t_5 does not compromise the validity of the output in t_{10} and t_{12}; otherwise, a precedence constraint between t_{11} and t_5 would exist.

7 Experimental Analysis

We split the evaluation part in three parts. The first one considers synthetic flows in a real environment. We use synthetic flows in order to thoroughly compare the techniques in a wide and configurable range of settings. In the second part, we focus on the computed cost of the resulting execution plans, which can be safely performed offline. The last part deals with the time overhead of the optimization techniques. According to [27], large flows are those that comprise 100 tasks, thus most of our experiments consider flows up to this size.

7.1 Data flows in a Real System

In this set of experiments, we present running times when executing synthetic flows in Pentaho Data Integration - Kettle (*PDI*, v. 5.2). PDI supports two execution modes, pipelined execution (default) and sequential. We chose the latter so that the measured running time corresponds to the sum cost metric (*SCM*) targeted in our algorithms. The machine we used is equipped with an Intel Core i5 660 CPU and 6 GB of RAM.

The main purpose is to evaluate the performance optimization, which corresponds to the minimization of the estimated flow execution cost *SCM*. The performance improvements of our algorithms are measured as either the decrease in running time or the speed-up achieved. The speed-up of a faster algorithm A with respect to a slower one B is defined as follows: $Speed-up = \frac{SCM(B)}{SCM(A)}$.

We construct synthetic flows so that we thoroughly evaluate the algorithms in a wide range of parameter combinations and we are in a position to derive unbiased and generically applicable lessons for the behaviour of each algorithm. The main configurable parameters are two: (i) the cost and selectivity values of the flow tasks, which are distributed in the range of $[1, 100]$ and $(0, 2]$, respectively; and (ii) the values of *DoF*, where we considered *DoF=0.2,0.4,0.6,0.8*; the smaller the *DoF* value, the less the opportunities for optimization exist. In this experiment, the size of the flows (i.e., the number of the tasks) is fixed to 30 and

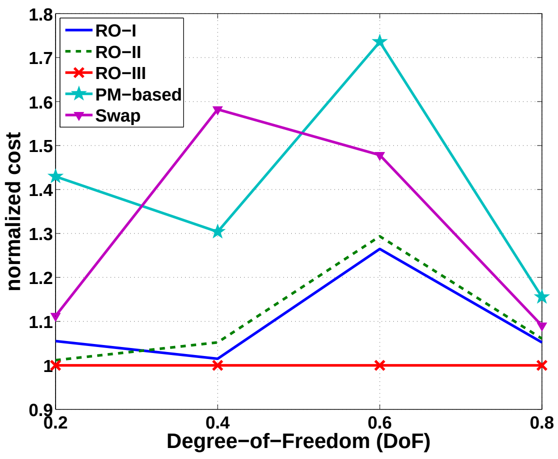


Fig. 15. Median normalized running times of a data flow with 30 tasks (the values are in normalized time units).

they process 100K records. In order to conduct the experiments, we randomly generate precedence constraint DAGs and task characteristics in a simulation environment. In PDI, all the tasks were dummy ones, i.e., they did not perform any actual processing, but they repeated a processes a number of iterations proportional to their cost and produced an output record according to their assigned selectivity.

For every *DoF* value, we generate 10 test PC DAGs, and each individual DAG instantiation is executed 5 times. Unless otherwise mentioned, median values are presented. The medians better represent the practical value, and in general are lower than the average values, which are affected by outliers.

Figure 15 shows the behavior of the rank ordering-based optimization techniques proposed in Section 4. In the figure, the running times are normalized according to the lowest ones achieved by RO-III, so that improvements are more clearly presented. We see that the highest median improvements over the best performing between *Swap* and *PM-based* are for *DoF* values 0.4 and 0.6. More specifically, *Swap* exhibits higher times by 47.8% and 58.2% for *DoF* values of 0.4 and 0.6, respectively. *PM-based* exhibits higher times by 30.3% and 73.6% for *DoF* values of 0.4 and 0.6, respectively. The improvements are lower for flows with *DoF* of 0.2 or 0.8. However, both these cases are more rare in practice: when *DoF*=0.8, simply ranking by the rank value as in filter ordering in database queries is sufficient, whereas, when *DoF*=0.2, there is relatively small space for improvements. In other words, when *DoF* is low there is small space for re-orderings, whereas, for high *DoF* values, the need for sophisticated constraint-aware algorithms is ameliorated.

Table 1. Maximum observed times RO-III is faster

alg* \ *DoF	**0.2**	**0.4**	**0.6**	**0.8**
PM-based	1.84	2.41	4.77	2.06
Swap	1.84	3.30	3.62	1.39

Regarding the behavior of our algorithms, *RO-I* outperforms *RO-II* by a small factor, for *DoF* values greater than 0.2. Also, the supremacy of *RO-III* is more evident for *DoF*=0.6.

The numbers thus far discussed are median values. However, in individual DAGs, we observed significantly larger improvements due to *RO-III.* The maximum improvements of *RO-III* are presented in Table 1, and as shown, for these 30-task flows executed in PDI, *RO-III* can run up to more than 4 times faster.

7.2 Synthetic Scenarios

In this section we present a more extensive set of experiments, where the size of flows n ranges from 10 to 100 (without including the source and sink tasks). Each combination of *DoF*, costs, selectivities and flow size is instantiated 100 times. In Section 2, we presented the clear gap between the best heuristic to date, namely *Swap*, and the accurate solution for a real small flow. We aim to show how the rank ordering-based solutions are capable to significantly improve the performance of the data flow execution and how the parallelism of *L-SISO* flows can be beneficial. Finally, we evaluate the proposals for *MIMO* flows.

Performance of Rank Ordering-based Solutions Our first experiment compares the techniques for *L-SISO* flows, and extends the evaluation rationale of the previous section in the sense that we include the performance of an initial non-optimized flow, derived by simple topological ordering of the constraints. Figure 16 presents the median speed-up achieved by the optimization solutions compared to the non-optimized case.

Several observations can be drawn. First, *RO-III* is a clear winner and its median performance is better in all cases without a single exception. Second, *PM-based* solutions achieve significantly lower speed-ups than the other optimization algorithms. This supports our observation that using rank values only is not sufficient. Third, the median improvements of *RO-III* over the best heuristic of the state-of-the-art *Swap* and *PM-based* can be significant, as the *RO-III* can have 3 times better performance than these heuristics; this difference is observed for $n = 90$ and $DoF = 0.8$ and for $n = 100$ and $DoF = 0.8, 0.4$. We compare *RO-III* against *Swap* and *PM-based* more thoroughly later. Fourth, *RO-I* outperforms *RO-II* for *DoF=0.2* on average, while *RO-II* outperforms *RO-I* for *DoF=0.8*. For *DoF=0.6,0.4*, there is not a clear winner between *RO-I* and *RO-II.* Finally, we see that *RO-II* behaves worse than *RO-III*, which implies that the extensions of the latter are effective.

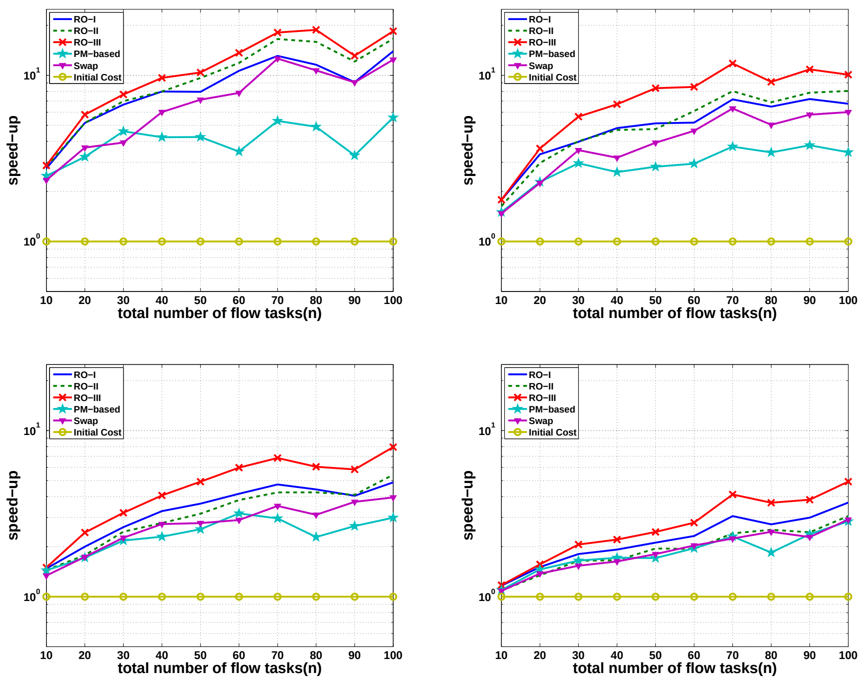


Fig. 16. Speed-ups with regards to the non-optimized flow for *DoF=0.8* (top-left), for *DoF=0.6* (top-right), for *DoF=0.4* (bottom-left) and for *DoF=0.2* (bottom-right).

When considering average instead of median execution times, the average speed-ups for *RO-III* reach three orders of magnitude. This can be explained with isolated runs, where the speed-ups are five or six orders of magnitude. In other words, the plots in Figure 16 are rather conservative in terms of potential of our proposals for improving on the SCM.

The numbers mentioned above refer to all the 100 flows for each combination of parameters. In Table 2, we present the speed-up that *RO-III* yields with regards to the best performing technique between *PM-based* or *Swap* on a more detailed basis (in each flow, a different technique may yield the highest performance). The table shows both the number of occurrences that one algorithm outperforms the others in each set of 100 random flows. For small flows of 10 tasks, there is a probability that all solutions yield the same execution plan. This probability ranges from 16% to 59% for flows with *DoF* equal to 0.8 and 0.2, respectively. For larger flows, it is extremely rare *PM-based* or *Swap* to find the same or a better plan (less than 1% of the cases).

The table also shows the average and median speed-ups. For flows with 100 tasks, the SCM drops to the half on average, if not more. For example, when $n = 100$ and $DoF = 0.4$, *PM-based* (resp. *Swap*) runs 11.35 (resp. 3.6) times slower than *RO-III* on average. In isolated runs, the performance improvements

Table 2. Detailed comparison of *RO-III* against the best performing between *Swap* and *PM-based*: number of cases, average and median speed-ups.

DoF=0.8							
	RO-III better			*same*	*RO-III worse*		
n	*♯times*	*avg*	*median*	*♯times*	*♯times*	*avg*	*median*
10	84	1.2895	1.1238	16	0	-	-
20	99	1.6539	1.3200	0	1	1.0656	1.0656
40	100	2.2130	1.3243	0	0	-	-
60	100	2.7779	1.4920	0	0	-	-
80	100	2.2587	1.2209	0	0	-	-
100	100	3.0435	1.1691	0	0	-	-
DoF=0.6							
	RO-III better			*same*	*RO-III worse*		
n	*♯times*	*avg*	*median*	*♯times*	*♯times*	*avg*	*median*
10	82	1.2981	1.1431	18	0	-	-
20	100	1.6657	1.2537	0	0	-	-
40	100	3.0637	1.4808	0	0	-	-
60	99	2.2811	1.4993	0	1	1.0103	1.0103
80	100	2.7548	1.8658	0	0	-	-
100	99	2.1876	1.3618	1	0	-	-
DoF=0.4							
	RO-III better			*same*	*RO-III worse*		
n	*♯times*	*avg*	*median*	*♯times*	*♯times*	*avg*	*median*
10	47	1.1993	1.1081	53	0	-	-
20	96	1.4283	1.2477	1	3	1.2489	1.0817
40	100	1.8940	1.3474	0	0	-	-
60	99	2.0278	1.4952	0	1	1.0909	1.0909
80	100	2.2472	1.5798	0	0	-	-
100	99	3.5996	1.5130	0	1	1.0793	1.0793
DoF=0.2							
	RO-III better			*same*	*RO-III worse*		
n	*♯times*	*avg*	*median*	*♯times*	*♯times*	*avg*	*median*
10	41	1.1375	1.0520	59	0	-	-
20	81	1.1495	1.0431	17	2	1.0471	1.0471
40	99	1.3974	1.1899	0	1	1.004	1.0471
60	100	1.5893	1.3329	0	0	-	-
80	99	1.6200	1.3645	0	1	1.0047	1.0471
100	100	1.9962	1.4778	0	0	-	-

are more impressive and reach two orders of magnitude. For example, we have observed speed-ups of up to 645 (resp. 98) times with regards to *PM-based* (resp. *Swap*) for n=100. For smaller flows of $n = 50$, the maximum observed speed-up is 76 (resp. 41) times.

We conclude this part of the discussion with a comment on *RO-III* vs. *RO-I*. In average, *RO-III* is more efficient than *RO-I*, but in 1-6% of the cases, it produces costlier plans than *RO-I* by up to 44%. Also, in some combinations of parameters, the plans may be the same. By design, *RO-III* is never worse than *RO-II*.

Performance of Parallel Optimization Solutions This set of experiments is conducted in order to evaluate the performance of data flows when they are executed in parallel according to the techniques discussed in Section 5. To this end, we compare the parallel version of *Swap*, named as *PSwap*, against the parallel

Table 3. Normalized SCM for data flows with n=50,100 tasks.

n=50				
alg\DoF	***0.8***	***0.6***	***0.4***	***0.2***
Initial	14.8634	10.6080	6.0250	2.6482
PSwap	1.3871	1.7109	1.4704	1.1854
PSwap′	1.4139	1.7389	1.5841	1.2188
PPM-based	2.5108	2.1285	1.6159	1.2030
PPM-based′	2.5813	2.2108	1.6600	1.2215
PRO-I	1.0985	1.2902	1.1688	1.0571
PRO-I′	1.1082	1.3011	1.1876	1.0748
PRO-II	1.0488	1.1814	1.3028	1.1930
PRO-II′	1.0538	1.1984	1.3251	1.2368
PRO-III	1.0000	1.0000	1.0000	1.0000
PRO-III′	1.0000	1.0002	1.0015	1.0037
n=100				
alg\DoF	***0.8***	***0.6***	***0.4***	***0.2***
Initial	37.8602	25.3375	18.3169	7.2005
PSwap	1.7214	2.1954	2.1684	1.8378
PSwap′	1.7778	2.2805	2.2565	1.9894
PPM-based	4.8242	4.4072	2.6290	1.8643
PPM-based′	4.8421	4.5622	2.7421	1.9683
PRO-I	1.5256	1.6575	1.4910	1.4188
PRO-I′	1.5410	1.7268	1.5312	1.5068
PRO-II	1.1097	1.5330	1.9290	1.9040
PRO-II′	1.1146	1.5670	2.0122	1.9817
PRO-III	1.0000	1.0000	1.0000	1.0000
PRO-III′	1.0000	1.0001	1.0009	1.0181

Table 4. Times per 100 runs where non-linear plans improve >10%

	n=50				***n=100***			
alg\DoF	***0.2***	***0.4***	***0.6***	***0.8***	***0.2***	***0.4***	***0.6***	***0.8***
PM-based	7	11	11	10	17	23	25	21
Swap	13	19	20	15	31	37	35	33
RO-I	9	6	10	10	15	17	19	12
RO-III	0	0	2	5	0	3	1	2

proposed rank ordering-based algorithms, denoted as *PRO-I,PRO-II,PRO-III*, respectively. We also compare against *PPM-based*. Initially, we assume that the merge cost mc is 0 and then, we repeat the experiments considering with non-zero merge cost. The value of the extra merge cost was defined after experiments in real data flows with the PDI tool. So, we set the $mc = 10$, that is an order of magnitude higher than the less expensive tasks and an order of magnitude lower than the most expensive ones. We examine data flows consisting of 50 and 100 tasks.

The comparisons are presented in Table 3, where it is shown that the parallelized version of *RO-III*, *PRO-III*, strengthens its position as the best performing technique. When the merge cost is considered, the names of the algorithms are coupled with the prime symbol (e.g., *PSwap*′); for the moment we do not focus on those table rows.

The aim of this evaluation is to show how often and to what extend parallelization is beneficial. The observed speed-ups are strongly correlated with the *DoF*. For low *DoF=0.2*, we observe that *PRO-III* has at least 19% lower execution cost than *PSwap* when the flow consists of 50 tasks. The performance improvement increases as the size of the flow and/or *DoF* increases. For example, when we optimize flows with 100 tasks for the same *DoF* the observed improvement is 84% and 86% against *Swap* and *PM-based*, respectively. Additionally, for flows with 100 tasks but higher *DoF=0.6*, we observe speed-ups up to 4 times.

The key observation after analyzing the individual runs is that, in the majority of the cases, parallel execution is beneficial. In the worst case, there is no performance improvement, producing non-linear plans can never lead to performance degradation. Table 4 shows the number of cases in each set of 100 examples, where the improvement was over 10%. We can see that *PSwap* is the algorithm that benefits the most from non-linear plans, whereas there are small benefits for *PRO-III*. This is partially due to the fact that *RO-III* already produces efficient plan what are harder to further improve upon.

We conclude that further refining the linear orderings with our proposed light-weight post-processing step can yield tangible performance improvements. The results after the application of the extra merge cost prove that its impact is not significant (see bottom part in Table 3) and the above observations still hold.

Performance of *MIMO* flows This set of experiments considers the evaluation of the methodology that is analyzed in Section 6 for *MIMO* data flow optimization. We consider two cases of butterfly flows (see Figure 12(left)). In each case we consider 10 linear segments with 10 and 20 tasks, respectively; thus the overall number of tasks is 100 and 200. The *DoF* of each linear segment is 0.6.

Figure 17 presents the median speed-ups of the *PRO-III*, *Swap* and *PM-based* algorithms over the non-optimized initial data flow. In the case where the linear segments are very small (10 tasks) the improvements are up to 86% (*PRO-III*). When the linear segment size increases to 20, *PRO-III* has median speed-up 3.8 times and 62% and 55% better performance improvement than *Swap* and *PM-based*, respectively. Overall, the results of the previous sections focusing on *L-SISO* flows generalize to *MIMO* flows as well.

7.3 Time Overhead

We also conduct an evaluation of the time overhead of the *RO-III*, *PRO-III* and *Swap* optimization algorithms. The purpose of this set of experiment is to

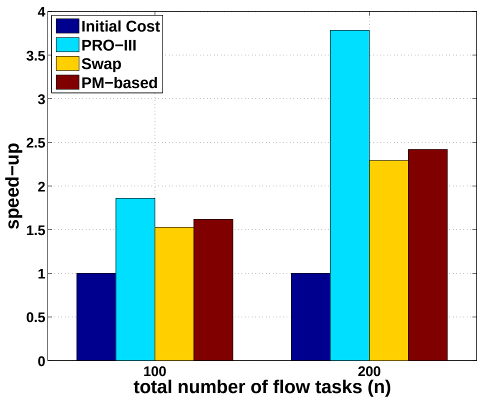


Fig. 17. *MIMO* optimization for n=100,200 and *DoF=0.6* for its linear segments.

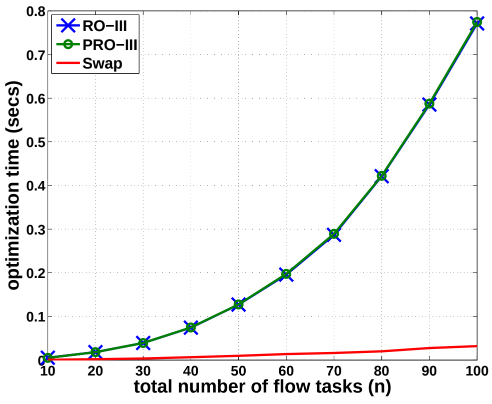


Fig. 18. The optimization time of *RO-III*, *PRO-III* and *Swap* for *DoF=0.6*.

show how these techniques scale as the size of data flow increases from 10 to 100 tasks for *DoF=0.6*. The findings show that the optimization time of *RO-III* and *PRO-III* algorithms is similar and their differences are negligible. In Figure 18, the key conclusion is that even for large flows, our proposals are capable of optimize a data flow in less than a second despite the fact that *Swap* algorithm scales better than rank ordering algorithms. The overhead of *RO-I* and *RO-II* lies in between the overhead of *RO-III* and *Swap* (not shown in the figure). Overall, our proposal seems to trade a very small time overhead for significant improvements in SCM.

8 Related Work

In this section, we take a deeper look at the proposals that optimize the flow execution plan through changes in the structure of the flow graph including task re-ordering.

For completeness, we start from flow optimization solutions that are inspired by query processing techniques, which are inferior to those already discussed in the beginning of Section 4. In [9], an optimization algorithm for query plans with dependency constraints between algebraic operators is presented. The adaptation of this algorithm in our *L-SISO* problem setting is reduced to optimization algorithms that are less efficient than *Swap* [17]. In [15], ad-hoc query optimization methodologies are employed in order to perform structure reformations, such as re-ordering and introducing new services in an existing workflow; in this work we investigate more systematic approaches.

Optimizations of Extract Transform Loading (ETL) flows are analyzed in [25]. In this proposal, ETL execution plans are considered as states and transitions, such as swap, merge, split, distribute and so on, are used to generate new states in order to navigate through the state space, which corresponds to the execution plan alternatives. However, the complete proposal for reducing the ETL workflow has exponential complexity. In our work, where we deal with task re-orderings only, the relevant part of the proposal in [25] corresponds to the *Swap* algorithm, which is explicitly considered in our evaluation. Additional simple heuristics proposed for minimizing the SCM have appeared in [20] and [34]; those heuristics are also reported to be inferior to *Swap* [17].

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. An extension has appeared in [24], but this solution is not scalable.

In addition, there is a significant portion of proposals on flow optimization that proceed to flow structure optimizations but do not perform task re-ordering, as we do. As such, they are orthogonal to our work. For example, the proposals in [33, 30] fall into this category. Several optimizations in workflows are also discussed in [4], but the techniques are limited to straightforward application of query optimization techniques, such as join re-ordering and pushing down selections. Additionally, there are optimization proposals for the parallel execution of online Web-Services represented as queries, such as the proposal in [28], which however aims to minimize the bottleneck cost rather than the sum of the task costs. The optimization techniques that have been proposed in [2] and [29] also aim to minimize the bottleneck cost. Another optimization proposal that targets a different optimization metric, namely throughput maximization in a parallel setting, is presented in [8]; the distinctive feature of this proposal is that it provides a set of concurrent execution plans. Finally, numerous proposals target efficient resource allocation. Contrary to our work, they assume an execution setting with multiple execution engines and do not deal with optimization of the flow task ordering, e.g., [18, 26]. [21] discusses methodologies about how to execute and dispatch task activities in parallel computers, while some other proposals deal with task scheduling, e.g., [3].

9 Conclusions

In this work, we deal with the problem of specifying the optimal execution order of constituent tasks of a data flow in order to minimize the sum of the task execution costs. We are motivated by the significant limitations of fully-automated optimization solutions for data flows with arbitrary tasks, as, nowadays, the optimization of complex data flows is either left to the flow designers and is a manual procedure or relies on solutions that have limited efficiency or scalability. To fill the gap of efficient optimization techniques, we initially focus on flows with a single data source and sink, and propose a set of approximate algorithms that can yield improvements of several factors on average, and several orders of magnitude in isolated cases. We then introduce a post-process optimization phase for parallel execution of the flow tasks to further improve the performance of a data flow. Finally, we show that we can extend these solutions to more complex data flow scenarios that deal with arbitrary number of sources and sinks.

There are several avenues for further research, including deeper investigation of optimizing arbitrary *MIMO* flows, and consideration of other types of constraints, e.g., not allowing two tasks to be placed in the same branch. A more ambitious goal is to provide more holistic flow optimization algorithms, which combine task ordering with aspects, such as task implementation and scheduling.

References

1. Pentaho - Data Integration (Kettle). `http://kettle.pentaho.com` (2014)
2. Agrawal, K., Benoit, A., Dufossé, F., Robert, Y.: Mapping filtering streaming applications. Algorithmica 62(1-2), 258–308 (2012)
3. Agrawal, K., Benoit, A., Magnan, L., Robert, Y.: Scheduling algorithms for linear workflow optimization. In: IPDPS (2010)
4. Böhm, M.: Cost-based optimization of integration flows. Ph.D. thesis (2011)
5. Burge, J., Munagala, K., Srivastava, U.: Ordering pipelined query operators with precedence constraints. Technical Report 2005-40, Stanford InfoLab (2005)
6. Chaudhuri, S., Dayal, U., Narasayya, V.: An overview of business intelligence technology. Commun. ACM 54, 88–98 (2011)
7. Chaudhuri, S., Shim, K.: Optimization of queries with user-defined predicates. ACM Trans. Database Syst. 24(2), 177–228 (1999)
8. Deshpande, A., Hellerstein, L.: Parallel pipelined filter ordering with precedence constraints. ACM Transactions on Algorithms 8(4), 41:1–41:38 (Oct 2012)
9. Florescu, D., Levy, A., Manolescu, I., Suciu, D.: Query optimization in the presence of limited access patterns. In: ACM SIGMOD
10. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database systems - the complete book (2. ed.). Pearson Education (2009)
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 work ows 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: DOLAP. pp. 95–98 (2014)
17. Kougka, G., Gounaris, A.: Cost optimization of data flows based on task reordering. CoRR abs/1507.08492 (2015)
18. Kougka, G., Gounaris, A., Tsichlas, K.: Practical algorithms for execution engine selection in data flows. Future Generation Computer Systems 45(0), 133 – 148 (2015)
19. Krishnamurthy, R., Boral, H., Zaniolo, C.: Optimization of nonrecursive queries. In: VLDB. pp. 128–137 (1986)
20. Kumar, N., Kumar, P.S.: An efficient heuristic for logical optimization of etl workflows. In: BIRTE. pp. 68–83 (2010)
21. Ogasawara, E.S., de Oliveira, D., Valduriez, P., Dias, J., Porto, F., Mattoso, M.: An algebraic approach for data-centric scientific workflows. PVLDB 4, 1328–1339 (2011)
22. Olston, C., Chopra, S., Srivastava, U.: Generating example data for dataflow programs. In: ACM SIGMOD
23. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: SIGMOD Conference. pp. 1099–1110 (2008)
24. Rheinländer, A., Heise, A., Hueske, F., Leser, U., Naumann, F.: SOFA: an extensible logical optimizer for udf-heavy data flows. Inf. Syst. 52, 96–125 (2015)
25. Simitsis, A., Vassiliadis, P., Sellis, T.K.: State-space optimization of etl workflows. IEEE TKDE 17(10), 1404–1419 (2005)
26. Simitsis, A., Wilkinson, K., Castellanos, M., Dayal, U.: Optimizing analytic data flows for multiple execution engines. In: ACM SIGMOD. pp. 829–840 (2012)
27. Simitsis, A., Wilkinson, K., Dayal, U., Castellanos, M.: Optimizing etl workflows for fault-tolerance. In: ICDE. pp. 385–396 (2010)
28. Srivastava, U., Munagala, K., Widom, J., Motwani, R.: Query optimization over web services. In: Proc. of the 32nd Int. Conference on Very large data bases VLDB. pp. 355–366 (2006)
29. Tsamoura, E., Gounaris, A., Manolopoulos, Y.: Decentralized execution of linear workflows over web services. Future Gener. Comput. Syst. 27(3), 341–347 (2011)
30. Tziovara, V., Vassiliadis, P., Simitsis, A.: Deciding the physical implementation of ETL workflows. In: DOLAP. pp. 49–56 (2007)
31. Van Der Aalst, W.M.P., Ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distrib. Parallel Databases 14(1), 5–51 (2003)
32. Vassiliadis, P., Karagiannis, A., Tziovara, V., Simitsis, A.: Towards a benchmark for ETL workflows. In: International Workshop on Quality in Databases, QDB. pp. 49–60 (2007)
33. Vrhovnik, M., Schwarz, H., Suhre, O., Mitschang, B., Markl, V., Maier, A., Kraft, T.: An approach to optimize data processing in business processes. In: VLDB. pp. 615–626 (2007)

34. Yerneni, R., Li, C., Ullman, J.D., Garcia-Molina, H.: Optimizing large join queries in mediation systems. In: ICDT. pp. 348–364 (1999)