Cost optimization of data flows based on task re-ordering

Georgia Kougka

*Aristotle University of Thessaloniki*

Anastasios Gounaris

*Aristotle University of Thessaloniki*

---

**Abstract**

Analyzing big data in a highly dynamic environment becomes more and more critical because of the increasingly need for end-to-end processing of this data. Modern data flows are quite complex and there are not efficient, cost-based, fully-automated, scalable optimization solutions that can facilitate flow designers. The state-of-the-art proposals fail to provide near optimal solutions even for simple data flows. To tackle this problem, we introduce a set of approximate algorithms for defining the execution order of the constituent tasks, in order to minimize the total execution cost of a data flow. We also present the advantages of the parallel execution of data flows. We validated our proposals in both a real tool and synthetic flows and the results show that we can achieve significant speed-ups, moving much closer to optimal solutions.

*Keywords:* data flows optimization, task reordering, PDI

---

1. **Introduction**

Data analysis in a highly dynamic environment 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, we typically employ

Email addresses: georkoug@csd.auth.gr (Georgia Kougka), gounaria@csd.auth.gr (Anastasios Gounaris)
fully automated data-centric flows (or simply called data flows) both for business intelligence [1] and scientific purposes [2], which typically execute under demanding performance requirements, e.g., to complete in a few seconds. Meeting such requirements, 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 [3]. Example steps include 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 [4]. The tasks in a flow can either have a direct correspondence to relational operators, such as filters, grouping, aggregates and joins, or encapsulate arbitrary data transformations, text analytics, machine learning algorithms and so on [5, 6, 3].

One of the most important steps in the data flow design is the specification of the execution order of the constituent tasks. In practice, this is usually the result of a manual procedure, which, in many cases results in non-optimal flow execution plans. Furthermore, even if a data flow is optimal for a specific input data set, it may prove significantly suboptimal for another data set with different characteristics [7]. We tackle this problem through the proposal of optimization algorithms that can provide the optimal execution order of the tasks in a data flow in an efficient manner and relieve the flow designers from the burden of selecting the task ordering on their own. We consider a single optimization objective, namely the minimization of the sum of the task execution costs; we assume that the execution cost of each task depends on the volume of data to be processed, which in turn depends on the relative position of the task in the execution flow. The main challenges in flow optimization that need to be addressed and differentiate the problem from that of traditional query optimization are as follows:

1. No arbitrary task orderings are valid, which means that the optimization algorithms need to respect the precedence constraints among tasks.
E.g., in the introductory example, we cannot move a task that computes the average sentiment value from tweets before executing the task that quantifies the sentiment of the user through text analysis.

2. Flows can be very large with many constituent tasks, e.g., up to one hundred.

The main implication is that query optimization techniques, which operate on plans with up to a few tens of operators that belong to the relational algebra (according to which operator reordering is typically permitted), are not applicable \[8, 9\]. Nevertheless, they are successful in their domain and this is the reason the data flow solutions proposed in this work are partially inspired by query optimization as we explain later. Overall, to date, there are very few proposals that deal with (or are applicable to) task reordering in data flows \[10, 11, 6\]. A common characteristic of these proposals is that they are too slow to find an exact solution in small flows \[6\], or they can find significantly suboptimal (approximate) solutions for bigger flows \[10, 11\].

In this work, we go beyond the state-of-the-art; we present both approximate and exact solutions. The approximate solutions are applicable to large flows and attain significantly better performance (more than 2 times speed-up in some settings, whereas in stand-alone cases, the speed-up is two or three orders of magnitude). The exact solution that we propose, although it cannot scale in general, it can process larger flows than those currently amenable to exact optimization. Our solutions apply to flows comprising any type of tasks and require as input common metadata that is task-independent, such as the average task selectivity and the task cost per invocation (e.g., in time units).

Initially, we target linear flows, that is flows that can be described as a chain of activities with a single source and a single sink task; later, we relax this assumption. 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 \[12\]. Additionally, we performed thorough evaluations against existing approaches for synthetic data flows. The summary of our contributions is as follows:

1. We provide a case study of data flow optimization implemented in PDI to provide insights into the inefficiency of the existing approaches and the actual benefits of our approaches (Section 3).

2. We show that, under certain conditions, it is practical to derive optimal linear flows, even when the number of tasks is relatively large. Contrary to the case of query optimization, the most efficient solutions are those
that leverage algorithms enumerating valid topological orderings rather than dynamic programming or backtracking techniques (Section 4).

3. 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 5).

4. We discuss algorithms that produce flow execution plans, where a task sends its output to several downstream tasks in parallel; such an approach is suitable when the task selectivities are above 1, and can further improve on the performance of the flow execution plans (Section 6).

5. We show how we can extend the solutions mentioned above to non-linear flows with arbitrary number of sources and sinks (Section 7).

6. We conduct thorough experiments in synthetic flows to detect the best optimization algorithm for linear and non-linear data flows among all of our proposals (Section 8). The evaluation results prove that the approaches introduced here significantly and consistently outperform the state-of-the-art in all out experiments.

An extended abstract of some of the ideas above appears in [13].

2. Problem Statement and Background

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 performance of the flow is maximized. 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; i.e., in data flows, the exchange of data between tasks is explicitly represented through edges. The main notation, terminology and assumptions are as follows:

- Let $G = (T, E)$ be a directed acyclic graph, 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, since it defines the exact execution order of the tasks.
• $T = \{t_1, \ldots, t_n\}$ is a set of tasks 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.

• Let $E = \{\text{edge}_1, \ldots, \text{edge}_m\}$ be a set of edges of size $m$. Each edge $\text{edge}_i, 1 \leq i \leq m$ equals to an ordered pair $(t_j, t_k)$ denoting that task $t_j$ sends data to task $t_k$. $m \leq \frac{n(n-1)}{2}$; otherwise $G$ cannot be acyclic.

• Let $PC = (T', D)$ be another directed acyclic graph, where $T' \subseteq T$. $D$ defines the precedence constraints (dependencies) that might exist between pairs of tasks in $T'$. More formally, $D = \{d_1, \ldots, d_l\}$ 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 $G$. In other words, $G$ should contain a path from $t_j$ to $t_k$. This implies that if $D$ contains $(t_a, t_b)$ and $(t_b, t_c)$, it must also contain $(t_a, t_c)$. 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.

• Two execution plans $G_1$ and $G_2$ that respect all the precedence constraints in $PC$ are termed as logically equivalent flows.

In this work we initially focus on single-input single-output (SISO) flows. A SISO data flow is defined as a flow $G$ that contains only one task with no incoming edges from another task and only one task with no outgoing edges. The task with no incoming edges is termed as the source task and the task with no outgoing edges is termed as the sink task. 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.

Examples of SISO flows are given in Figure 1. In the figure, we can see that a SISO flow can be executed both as a linear flow and as a parallel flow. In linear physical 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 parallel physical flows, the output of a single task can be fed to multiple tasks in parallel. The linear flow and the parallel flows in the figure are logically

\[\text{In the remainder of the paper, we will use the terms tasks, services and activities interchangeably.}\]
equivalent flows. Because each SISO flow is logically equivalent to at least one linear $G$, we call SISO flows as logically (or conceptually) linear flows.

Each task is further described as a triple $t_i = <c_i, sel_i, inp_i>$. In a dataflow, 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 elements are:

- **Cost** ($c_i$): we use $c_i = 1/r_i, 1 \leq i \leq n$ as a metric of the time cost of each task, where $r_i$ is the maximum rate at which results of invocations can be obtained from the $i$-th task.

- **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, $sel_i > 1$.

- **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 $G$. More formally, if $T_i^{\text{prec}}$ is the set of all preceding tasks of $t_i$ in $G$, $inp_i = \prod_{j=1}^{|T_i^{\text{prec}}|} sel_j$.

- **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 $inp_i sel_i$.

From the above quantities, and assuming that selectivities are independent, we can infer that $inp_i$ is the only task characteristic that depends on

---

2Here, 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.
the position of \( t_i \) in \( G \); the cost and the selectivity of each task is independent of the exact \( G \) that may include \( t_i \).

**Problem Statement:** Given 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: \( SCM(G) = \text{inp}_1c_1 + \text{inp}_2c_2 + \ldots + \text{inp}_nc_n \). The optimal plan is denoted as \( P \).

Note that the input set of tuples are processed by all the tasks of the data flow, but typically, some of the input tuple attributes may not be required by every flow activity. According to [14], the unnecessary tuple attributes just run through the flow, resembling an assembly-line model. The execution of a flow activity is not affected by the unnecessary attributes. This implies that the tasks of a flow have the ability to be reordered as long as the precedence constraints between the tasks are preserved.

### 2.1. Problem Complexity

In [15] 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. Moreover, in [13] 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 \( \theta \) is some positive constant. Note that if we modify the optimization criterion, e.g., to optimize the bottleneck cost metric or the critical path renders the problem tractable [16, 17].

### 3. Motivational Case Study in Kettle

In this section, we present the application of data flows in a real-world business tool, named as Pentaho Data Integration (PDI) (Kettle) [12], in order to highlight the impact of optimization proposals in the performance of a flow execution. We introduce a data flow (Figure 2) that analyzes tags referring to products, which are retrieved from tweets in Twitter, in order to compose a dynamic report that associates sales with marketing campaigns. In the following, we analyze the tasks of this data flow and of a flavour of it
combined by details of the data set that the data flow process for the case study purposes.

As we observe this data flow has a single streaming source that outputs tweets on products and the flow accesses four other static sources through lookup operations. The initial streaming source task, called as Tweets, of the flow consists of 1,000,000 records of tweets with attributes, such as product references, coordinates, timestamps etc. More specifically, the data flow is described as follows. When a tweet arrives as a timestamped string attribute (tag), the first task is to compute a single sentiment value in the range [-5 5] for the product mentioned in the tweet (Sentiment Analysis). Then, a lookup operation which maps product references in the tweet is performed (Lookup ProductID) and after this a filter is applied in order to choose products with a specified range of product id values (Filter products). The next task is also a lookup task which maps geographic information (latitude and longitude) in the tweet to a geographical region (Lookup Region). In the following, the task Extract date from timestamp converts the tweet timestamp to a date and then, another filter is applied for choosing dates for a specific period of time (Filter Dates). In order to implement the task SentimentAvg, where the sentiment values are averaged over each region, product, and date, we first have to sort the values of region, product, and date by applying the task Sort Region, Product and Date. The flow continues with other two lookup operations: the former maps the total sales of a product by the region, product and date (Lookup Total Sales) and the latter maps campaigns of interest according to the results of total sales taken from the previous task.

Figure 2: A real-world analytic flow.
Table 1: The cost and selectivity values.

| ID | Flow Task                          | Cost(secs) | Selectivity |
|----|------------------------------------|------------|-------------|
| 1  | Tweets (data source)               | 1.7        | 1           |
| 2  | Sentiment Analysis                 | 4.5        | 1           |
| 3  | Lookup ProductID                   | 5          | 1           |
| 4  | Filter Products                    | 1.9        | 0.9         |
| 5  | Lookup Region                      | 6.5        | 1           |
| 6  | Extract Date from Timestamp        | 19.4       | 1           |
| 7  | Filter Dates                       | 2          | 0.2         |
| 8  | Sort Region, Product and Date      | 17.3       | 1           |
| 9  | SentimentAvg                       | 10.3       | 0.1         |
| 10 | Lookup Total Sales                 | 10.8       | 1           |
| 11 | Lookup Campaign                    | 11.6       | 1           |
| 12 | Filter Region                      | 2          | 0.22        |
| 13 | Report Output                      | 1          | 1           |

(Lookup Campaign). Finally, the user has the option to narrow down the report in order to focus on a specific region with the filtering task Filter Region.

Additionally, there are four intermediate static sources, used as inputs in lookup operations, whose cost is embedded in the cost of the task where the static records are taken as inputs of the lookup task executions. The source task Products has 100 records of product names and ids, while that Region source task has 100 records of set of coordinates corresponding to a region name. Another source static task named Sales consists of 4,000 sale details, such as the sold product name, the price, the quantity, the region where the product was sold etc., and the last one static source task, named Campaings, has 500 campaign ids combined with the day that these campaigns begin, the region that will take place, but also the product ids that each campaign concern.

Table 1 shows the selectivity and cost values computed for a specific dataset of 1M records using a machine with an Intel Pentium G860 CPU and 4 GB of RAM. We can observe that the most expensive tasks are the grouping and lookup tasks, the cost of which is up to two orders of magnitude compared to the less expensive ones. Also, there are three filtering tasks, while the rest of the tasks do not modify the number of records (note that in general, selectivities may be higher than 1). In this data flow scenario, the selectivity values of the lookup and transformation tasks is 1, while the selectivity values corresponding to filtering and grouping tasks varies.

In Table 2 the precedence constraints that tasks have between them are presented, having in the left part of the arrows the tasks that must precede the tasks that are defined in the right part of the table. This data flow...
Table 2: The precedence constraints of the data flow in Figure.

| Precedence Constraints |
|------------------------|
| SentimentAnalysis → SentimentAvg |
| LookupProductID → Filterproducts |
| LookupProductID → SortRegion, ProductandDate |
| LookupProductID → LookupTotalSales |
| LookupProductID → LookupCampaign |
| LookupRegion → SortRegion, ProductandDate |
| LookupRegion → LookupTotalSales |
| LookupRegion → LookupCampaign |
| LookupRegion → FilterRegion |
| Extractdatefromtimestamp → FilterDates |
| Extractdatefromtimestamp → SortRegion, ProductandDate |
| Extractdatefromtimestamp → LookupTotalSales |
| Extractdatefromtimestamp → LookupCampaign |
| SortRegion, ProductandDate → SentimentAvg |

has 38% precedence constraints, as they described in Table 2, where a fully constrained flow with \( n \) tasks and 100% PCs has \( \frac{n(n-1)}{2} \) constraints and no equivalent ordering alternatives. In real data flow scenarios the preserving precedence constraints are approximately 30% or even more, as the flows presented in [4].

A straight-forward implementation is shown in Figure 2. Then, we applied

Figure 3: The optimized plan by a heuristic algorithm of the flow in Figure 2

Figure 4: The optimized plan by an exhaustive algorithm of Figure of the flow in 2
best-performing approximate heuristic to date, which is proposed in [10].
The optimized plan is illustrated in Figure 3. In that case, the performance improvement from the initial non-optimized flow is 42% from 63 to 36.5 seconds.

Similar to the previous optimization, we applied our exhaustive solution to the flow of Figure 2 in order to find the optimal flow execution cost. In Figure 4, the optimal plan of the initial data flow is depicted. In this case, the exhaustive optimization methodology transposes the filtering task Filter Region, which at the initial design has been placed at the end as a final optional step, at the very beginning for this specific flow due to the metadata in Table 1. A less obvious optimization is to move the pair of date extraction and filtering tasks upstream although the former is expensive and not filtering. The execution cost of this optimized plan is 18.3 and results to a plan that is 3 times better than initial non-optimized. Both of the mentioned optimization methodologies are analyzed in the following sections.

This is a representative example of a real manually designed data flow that exhibits significantly suboptimal behavior. In general, we can draw two observations. Firstly, optimal solutions may yield lower execution costs by several factors. A second equally important observation is that even in simple cases like the one examined here, existing heuristics may fail to closely approximate the optimal solution and generate the plan in Figure 4. The main reason in this example is that the approximate solution performs greedy swaps of adjacent activities; however, the region filter cannot move earlier unless the campaign lookup task is moved earlier as well, an action that a greedy algorithm cannot cover.

4. Accurate Algorithms for Linear Execution Plans

In this section, we present three accurate algorithms for reordering SISO data flows in order to generate an optimal execution plan. The algorithms are based on backtracking, dynamic programming and generation of all topological sortings, respectively. Our main novelty here is that we examine a topological sorting-based algorithm, despite its worst-case complexity. Counter-intuitively, as we show in the evaluation, the algorithm is practical even for large $n$, when there are many precedence constraints and, in general, can scale better than the two other options. However, still, it cannot be applied to arbitrary flows of medium or large size.
4.1. Backtracking

The Backtracking algorithm finds all the possible execution plans generated after reordering the tasks of a given data flow preserving the precedence constraints. The algorithm enumerates all the valid sub-flow plans after applying a set of recursive calls on these sub-flows until generating all the possible data flow plans. It backtracks when a placement of a task in a specific position violates the precedence constraints. The algorithm is proposed for flow optimization in [6].

Complexity: The worst case time complexity of Backtracking is factorial (i.e., \(O(n!)\)), since, if there are no dependencies, all orderings will be examined in a brute force manner.

4.2. Dynamic programming

This algorithm is extensively used as part of the System R-type of query optimization to produce (linear) join orderings [18]. The rationale of the dynamic programming algorithm (termed as DP henceforth) for data flows remains the same, that is to calculate the cost of task subsets of size \(n\) based on subsets of size \(n-1\). For each of these subsets, we keep only the optimal solutions, which are valid with regards to the precedence constraints. Specifically, the DP algorithm considers each flow of size \(n\) as a flow of \((n-1)\) tasks followed by the \(n\)th task; the key point is that the former part is the optimal subset of size \(n-1\), which has been found from previous step; then the algorithm exhaustively examines which of the \(n\) flow tasks is the one that, when added at the end, yields an optimal subplan of size \(n\). For example, the algorithm starts by calculating subsets that consist of only one task \(\{t_1\}\), then \(\{t_2\}, \{t_3\}\) and so on. In a similar way, in the second step, it examines subsets containing two tasks, i.e., \(\{t_1, t_2\}, \{t_1, t_3\}\) and so on, until it examines the complete flow \(\{t_1, t_2, ..., t_n\}\). The number of the optimal (non-empty) subsets of a flow is equal to \(2^n - 1\). More details, along with pseudocode and an example are provided in Appendix A.

Complexity: The time complexity is \(O(n^22^n)\). This is because we examine all subsets of \(n\) tasks, which are \(O(2^n)\). For each subset, which is up to size \(O(n)\), we examine whether each element can be placed at the end of the subplan. Each such check involves testing whether any of the rest \(n-1\) tasks violate a precedence constraint, when placed before the \(n\)-th task. Overall, for each element, we make \(O(n)\) comparisons. So, the overall time complexity is \(O(2^n)O(n)O(n) = O(n^22^n)\). The space complexity is derived by the size of the auxiliary data structures employed. We use three vectors of size \(2^n - 1\)
as explained in Appendix A, the one of which stores elements of size $O(n)$. So the space complexity is $O(n^2n)$.

4.3. Topological sorting

The TopSort algorithm is a topological sorting algorithm based on [19], which finds all the possible topological sortings given a partial ordering of a finite set; in our case the partial ordering is due to the precedence constraints. The reason behind using this algorithm is that it (implicitly) prunes invalid plans very efficiently and it generates a new plan based on a previous plan after performing a minimal change. For the purposes of this work, we adapted the topological sorting algorithm in order to generate all the possible execution plans of a data flow and detect the execution plan with the minimum cost. The algorithm assumes that it can receive as input a valid task permutation $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow ... \rightarrow t_n$, which is trivial since it can be done in linear time. We generate all other valid execution plans by applying cyclic rotations and swapping adjacent tasks.

Firstly, the process of generating all the valid flow execution plans begins with the topological sorting of the $n-1$ tasks $t_2 \rightarrow t_3 \rightarrow ... \rightarrow t_n$ of the flow. Based on this partial sorting, we generate all the valid orderings of the $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow ... \rightarrow t_n$ plan. Specifically, in the first stage of the algorithm the task $t_1$ is placed on the left part of the partial plan $t_2 \rightarrow t_3 \rightarrow ... \rightarrow t_n$ and in the next steps of this stage, we swap it with the tasks on its right, while the tasks of the partial plan maintain their relative position. The $t_1$ stops moving when such a swap violates a precedence constraint. Then, as the task $t_1$ cannot be further transposed, the second stage of algorithm begins with a right-cyclic rotation of another partial plan consisted of $t_1$ and all the tasks that precede it, which means all the tasks which are positioned to its left. In this way, $t_1$ is placed to its initial position. Similarly, we generate all the topological sortings of $t_2 \rightarrow t_3 \rightarrow ... \rightarrow t_n$, $t_3 \rightarrow t_4 \rightarrow ... \rightarrow t_n$ and so on. For example, the topological sorting of $t_4 \rightarrow t_5 \rightarrow ... \rightarrow t_n$ partial plan will be generated with the transpositions of task $t_4$. For each generated plan, we estimate the total execution cost and finally, we choose the flow execution plan with the best performance. A pseudocode, a brief example and further details are in Appendix B.

Complexity: Since the algorithm checks all the permutations the time complexity is $O(n!)$ in the worst case. However, compared to other algorithms that produce all topological sortings, it is more efficient [19]. The space
complexity is $O(n)$ because only one plan is stored in main memory at any point of execution.

5. Approximate Algorithms for Linear Execution Plans

Due to the high complexity of the problem in hand, we need to develop approximate solutions for the generic case. This section consists of two parts: we first present existing solutions including straightforward extensions of existing proposals that are applicable to our problem, and then we present our main novelty with regards to approximate optimization of linear data flows. As will be shown in the evaluation, there is a significant gap in the performance between optimal solutions and existing approximate algorithms, and our proposal fills that gap.

5.1. Existing Solutions

Here we present four algorithms, which reflect the current state-of-the-art in task re-ordering in linear flows. Implementation details and examples are provided in Appendix C.

5.1.1. Swap

The Swap algorithm starts with a random valid execution plan. Such a plan is trivial to be computed in linear time through a single topological ordering of $PC$. The algorithm then compares the cost of the existing execution plan against the cost of the transformed plan, if we swap two adjacent tasks provided that the constraints are always satisfied. We perform this check for every pair of adjacent tasks and we repeat until no changes occur. Swap is equivalent to the proposal in [10] when only task re-ordering is allowed. The complexity of the Swap algorithm is $O(n^2)$ because we can repeat at most $n$ times, and each iteration has $O(n)$ complexity. The space complexity is linear ($O(n)$), equal to the complexity needed to store a single plan.

In order to prove that Swap is approximate, it is adequate to provide at least one example that the algorithm fails to yield the optimal solution. Assume a flow, which has three inner tasks (i.e., tasks other than the source and sink ones), each with cost equal to 1 and selectivities 1, 1.1, and 0.5 respectively. There is also a precedence constraint between tasks 2 and 3. If the initial plan is $t_1 \rightarrow t_2 \rightarrow t_3$, then its $SCM = 1 + 1 + 1.1 = 3.1$. However,

\[\text{An initial introduction of the existing algorithms has appeared in [20].}\]
the optimal plan is $t_2 \rightarrow t_3 \rightarrow t_1$ with $SCM = 1 + 1.1 + 0.55 = 2.65$. Swap cannot produce that plan because it cannot perform transpositions that initially produce worse plans, but eventually lead to better solutions, such as the swap of tasks $t_1$ and $t_2$.

5.1.2. GreedyI and GreedyII

GreedyI starts with an empty plan and in each step, it adds the activity with the maximum value of $(1 - sel_i)/(c_i)$, provided that it meets the precedence constraints. In the first step, the source task is chosen as the only eligible one. It bears similarities with the Chain algorithm in [11], although the latter algorithm was proposed for a different problem and appends the activity that minimizes $c_i$. The time complexity of Greedy algorithm is $O(n^2)$ because it consists of $n$ steps, where in each step $O(n)$ checks are performed to find the most efficient and valid task to append. With the help of appropriate data structures, the complexity can drop to $O(n \log n)$.

Similarly to Swap, it may miss the optimal solution. For example, in the example with the three tasks of cost 1 and selectivities 1, 1.1 and 0.5, and a precedence constraint between $t_2$ and $t_3$, GreedyI will first append $t_1$, then $t_2$ and last $t_3$, which is not the best possible plan as explained earlier.

Another greedy algorithm is GreedyII [21]. The rationale of GreedyII is similar to GreedyI apart from the fact that the construction of the optimized execution plan is right-to-left (i.e., from the sink to the source).

5.1.3. Partition

Partition forms clusters with activities by taking into consideration their eligibility. Specifically, each cluster consists of activities that their prerequisites have been considered in previous clusters. After building the clusters, each cluster is optimized separately by checking each permutation of cluster tasks. Similar to GreedyI, it was first proposed for data integration systems, and the details are given in [11]. Partition runs in $O(n!)$ time in the worst case because, if there are no precedence constraints, it checks all permutations of a partition of size $n$. In general, its complexity is $O(k!)$, where $k$ is the size of the largest cluster, and thus is inapplicable if a cluster contains more than a dozen of tasks. As in the previous optimality examples with the three tasks, it is easy to verify that it cannot find the optimal plan. In the first step, it forms a cluster with tasks $t_1$ and $t_2$ and decides to place $t_1$ before $t_2$ because is yields a better subplan.
5.2. Algorithms based on rank ordering

The motivation behind our proposal is that the approximate solutions discussed previously deviate significantly from the optimal orderings. To prove this, we conduct experiments with small flows, where applying an exhaustive technique to obtain the optimal plan is feasible. More specifically, in Figure 5 (left), we examine 100 randomly generated data flows consisting of 15 tasks with \( \text{cost} \in [1, 100] \), \( \text{sel} \in (0, 2] \) and 20%-95% precedence constraints. The results show that the performance improvement derived by the application of an accurate algorithm is high; see that \textit{TopSort} algorithm can have up to 57% better performance improvement compared to a random initial flow that just respects the precedence constraints. In general, \textit{Swap} seems to be the heuristic algorithm with the best performance improvement on average.

In Figure 5 (right), the maximum normalized difference between \textit{Swap} and \textit{TopSort} algorithms is presented. As we can observe, there are cases where the \textit{TopSort} algorithm has 74% better performance improvement than the best heuristic. These findings highlight the need for proposing new approximate optimization methodologies, in order to provide more near-optimal flow execution plans.
To fulfill this need, we propose a set of rank ordering-based approximate algorithms and we analyze them in this section. We build upon the join ordering algorithms proposed for query optimization in [22, 23], which will be referred to as KBZ. This algorithm leverages the rank value of each task defined as $\frac{1 - \text{sel}_i}{c_i}$ and the dependencies among tasks. Our solutions can be described at a high-level as shown in Algorithm 1. The main novelty is how to preprocess the flow, so that KBZ becomes applicable. Also, we post-process the result of the KBZ algorithm in order either to guarantee validity or to further improve the intermediate results. There are many options regarding how these two phases can be performed and here we present three concrete suggestions, which constitute the novelty of this section (examples are shown in Appendix D).

5.2.1. KBZ

The KBZ algorithm, which was proposed in [23], is a seminal query optimization algorithm for join ordering. This algorithm considers only a specific form of precedence constraints, namely those representable as a rooted tree. The rationale of this algorithm is to order tasks according to their rank value. In the case that this is not possible due to the defined precedence constraints, the tasks are merged and the rank values are updated accordingly. 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 percentage of precedence constraints is very low and decreases more with the number of tasks (e.g., less than 10% for a 100-node flow). Both of these cases do not occur frequently in practice. The time complexity of KBZ algorithm is $O(n^2)$.

5.2.2. RO-I

In our first proposal, called RO-I, the pre-processing phase ensures the transformation of the PC graph into a tree-shaped one. This is done by removing incoming edges with no maximum rank, if a task has more than one incoming edge. This allows KBZ to run but may produce invalid flow orderings. To fix that, we employ a post-processing phase where any resulting PC 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 ($O(n)$ complexity) 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 \( O(n^2) \). However, in practice the average time complexity is much lower for both phases.

5.2.3. RO-II

The RO-II algorithm follows a different approach in order to render KBZ applicable. 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. An example is shown in Figure 6. 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 \( t_5 \) but also tasks \( t_2 \) and \( t_4 \). 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.

RO-II, as will be shown in the evaluation section, in general behaves better than RO-I. However, in some cases RO-II’s performance is much worse and an example is in Appendix D. Also, 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 \( n \).
Algorithm 2 RO-III

Require: A set of n tasks, \( T = \{t_1, ..., t_n\} \)  
A directed acyclic graph \( PC \) with precedence constraints  
Optimized plan \( P \) 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
7:     end for
8:   end for
9: end for
10: until no changes applied

5.2.4. RO-III

After the evaluation of the proposed RO-I and RO-II algorithms, we isolated data flow cases that were not near-optimal. For example, the RO-II was not able to reorder a filtering task in an earlier stage of the flow, even when was not restricted by precedence constraints, in order to reduce the data that the flow will process. To fill this gap, we propose RO-III to support the efficient optimization of such data flow cases. The RO-III algorithm, presented in Algorithm 2, tackles the limitations of RO-II with the help of a post-processing phase that we introduce. Specifically, we 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 transpositions 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 (see the transposition of \( t_7 \) in Figure D.24 in Appendix D). This process is described by the 3 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 in practice, even for large flows, there is no change after 3 times. In all experiments, we set $k$ to 5.

6. Parallel Optimization Solutions

This section focuses on the advantages of parallel execution plans. As we have explained in Section 2, in a parallel physical 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 1, we observe that a single task may have not only multiple outgoing edges, but also multiple ingoing edges. In this case, a single task receives as input data the output of multiple tasks. This is in line with the AND-Join workflow pattern as presented in [24], where the outgoing edge of multiple tasks that are executed in parallel converge into a single task.

This case can be considered as a merge-split process, which in software tools such as PDI can be implemented by incorporating a merge join process. 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 cost; 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 2.
Let us now analyze when the parallel flow execution may be beneficial through a theoretical example. Let us consider two subsequent tasks $t_3$ and $t_4$ illustrated in Figure 7, which do not have precedence constraints between them 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) $inp_{linear}^{4} c_4 < inp_{parallel}^{4} c_4$ as $inp_{linear}^{4} = sel_3 inp_{parallel}^{4}$ and $sel_3 < 1$, and (ii) $inp_{linear}^{5} c_5 < inp_{parallel}^{5} (c_5 + mc)$ due to the extra merge cost of the parallel version and given that $inp_{linear}^{5} = inp_{parallel}^{5}$. 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 $inp_{linear}^{5} c_5 = inp_{parallel}^{5} (c_5 + mc)$. Because of the fact that $sel_3 > 1$, we deduce that $inp_{linear}^{4} c_4 > inp_{parallel}^{4} c_4$. 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 7. However, an optimized linear plan will place $t_4$ before $t_3$ thus corresponding to Case I, where the (new) linear plan is better than the parallel one.

As in the previous section, we first describe a simple extension to an existing solution for ordering web services described in [16]. Then we propose a novel post-processing step that applies to any of the solutions in the previous section in order to render their output plans into parallel ones. Our solution leverages and generalizes the analysis above, and based on the findings of Case III, it parallelizes tasks with selectivity higher than 1.
6.1. PGreedyI and PGreedyII

The *PGreedyI* optimization algorithm has the distinctive feature of generating parallel flow execution plans. The rationale of the *PGreedyI* is to order the flow tasks in such a way that the amount of data that is received by the tasks with selectivity value > 1 is reduced by pushing the selective flow tasks (filtering tasks) in an earlier stage of the flow to prune the input dataset. Based on the selectivity values, the optimal execution plan may dispatch the output of a task to multiple other tasks in parallel, or place them in a sequence. Specifically, the flow tasks having selectivity value > 1 are candidates for parallel execution in a flow. To this end, we employ the algorithm in [16] for generating parallel flow execution plans. The detailed description of the *PGreedyI* algorithm is presented in Appendix E.

A weak point of *PGreedyI* is that, in each step, it tries to find the task that has the minimum cost without considering the implications for the next tasks (e.g., due to high selectivity). The second flavour, *PGreedyII*, chooses not the activity with the less cost but the activity with the highest rank value; in this way we penalize tasks that have low cost but high selectivity, which can yield lower *SCM* values for the overall plan. Both algorithms have time complexity in $O(n^5)$ in the worst case, as explained in [16].

6.2. Executing SISO flows in parallel
**Algorithm 3** Post-process step for parallel SISO flows

**Require:** An optimized linear plan \( P = \{ t_1 \rightarrow ... \rightarrow t_n \} \)

A directed acyclic graph \( PC \) 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 \( PC \) for no task in \( t_{i+1} \ldots t_{j-1} \) 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 \( PC \) 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 \( t_{P(i+1)} \ldots t_{P(j-1)} \) with no outgoing edges in \( P \) and (ii) \( t_{P(j)} \)
14: \( i=j \)
15: end while

In order to exploit the advantages of the proposed optimization techniques, in Algorithm 3, we introduce a post-process phase for executing data flows in parallel. To this end, 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 to be 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.

An example is presented in Figure 8, where in the upper flow scenario, we choose to parallelize the tasks \( t_2, t_3 \) and \( t_4 \), while in the flow case that is depicted in the bottom of the figure, we execute parallel only the tasks \( t_2 \) 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 the task \( t_6 \) has selectivity value < 1, it is not executed in parallel with any other
The complexity is $O(n^2)$. The parallelization of each task is examined at most once, and each such case the preceding tasks need to be checked, the number of which cannot exceed $n$.

7. Extensions to MIMO flows

Algorithm 4 Optimization of MIMO flows

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

So far we have discussed the case with a 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 [25], two of which are shown in Figure 9. A main difference between SISO and MIMO flows is that apart from re-ordering tasks, additional optimization operations can apply. As explain in [10], 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. This allows for example a filtering operation initially placed after a merge task to be pushed down to the merge inputs (provided that

\[10\] additionally considers the case that an activity can be further split in several sub-activities, which is not considered here.
the filtering condition refers to both inputs), which is known to yield better performance.

As we can see in Figure 9, the MIMO flows consist of sub-linear flows. Therefore, the optimization of SISO data flows can play an important role in optimizing MIMO flows. Algorithm 4 describes a proposal for optimizing MIMO flows, which is based on the extraction of the linear segments of the flow and apply optimization algorithms only on the SISO sub-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.

8. Experimental Analysis

In this section we present a set of experiments, which have been conducted in order to evaluate the following two factors:

- **Performance** optimization, which corresponds to the minimization of the estimated flow execution cost \( SCM \). The performance improvements are measured as the percentage of the decrease in \( SCM \) after optimization.

- **Time Overhead**, in terms of real time that the generation of the optimized execution plan requires.

We construct synthetic flows so that we thoroughly evaluate the algorithms in a wide range of parameter combinations, so that we can derive unbiased and generically applicable lessons for the behaviour of each algorithm. The main configurable parameters are three: (i) the number of tasks \( n \) ranging from 10 up to 100 (without including the source and the sink tasks) thus covering a range from small to very large data flows; (ii) the cost and selectivity values of the flow tasks, which are distributed in the range of \([1, 100]\) and \((0, 2]\), respectively (following either the uniform or the beta distribution); and (iii) the number of precedence constraints between the flow tasks; in general we consider cases where there are \( \alpha \frac{n(n-1)}{2} \) constraints, where \( \alpha \in [0.1, 0.98] \). The larger the \( \alpha \) value, the less the opportunities for optimization exist. For small \( \alpha \) values, there are few PCs, which implies
the existence of several valid re-orderings. However, when $\alpha$ becomes 0, the problem reduces to filter ordering in database queries without precedence constraints and thus is out of our interest. Remember that in real cases, we expect PCs to be above 30%.

In order to conduct the experiments, we randomly generate PC DAGs and task characteristics in a simulation environment. Unless otherwise mentioned, every experiment is repeated 100 times and the average values are presented. When discussing real times, we use a machine with an Intel Core i5 660 CPU and 6 GB of RAM.

8.1. Performance Improvements

In the beginning of Section 5.2, we presented the significant gap between the best performing heuristics to date, namely Swap, and the accurate solutions for small flows. We extrapolate that this gap remains, if not widens, for larger flows. The main purpose of this part is to show how the rank ordering-based solutions are capable of filling this gap, and then we discuss the performance benefits due to parallelism in SISO flows. Finally, we evaluate the proposals for MIMO flows.

algorithms is presented. As we can observe, there are cases where the topSort algorithm has 74% better performance improvement than the best heuristic. These findings highlighted the need for proposing new approximate optimization methodologies, in order to provide more near-optimal flow execution plans.

8.1.1. Performance of Rank Ordering-based Solutions

Figure 10 presents the results of the comparison of rank ordering-based optimization methodologies with the initial flow execution plan and Swap. The values of the results are normalized according to the performance of the initial (random) execution plan. The four sub-figures present the performance improvement of each optimization proposal for $PCs = 20\%, 40\%, 60\%, 80\%$, respectively. Based on these results, a main observation is that RO-III is a clear winner, as it outperforms all the other optimization algorithms on average for all the PC percentages examined. The lesson is that the average improvements of RO-III over Swap can be significant, as the RO-III can yield up to 41% better performance than Swap on average; this difference is observed for $n = 80$ and $PC=40\%$, and means that RO-III is on average 1.69 times faster than Swap for that case. In addition, the maximum observed speed-up in isolated cases is much higher. For example, in one run where
Figure 10: Improvements in the SCM metric for PCs=20% (top-left), for PCs=40% (top-right), for PCs=60% (bottom-left) and for PCs=80% (bottom-right).

$n = 60$ and PC=60%, we have observed a speed-up of more than 73 times in favor of RO-III. In another run for $n = 100$ and PC=40%, the speed-up exceeded 285 times (two orders of magnitude).

RO-I seems to outperform RO-II for 80% precedence constraints on average, however, if we zoom on the isolated runs, in a significant portion of plans, RO-II is better. For less precedence constraints, there is not a clear winner between RO-I and RO-II.

Another significant observation from this figure, combined with Figure 5, is that RO-III eliminates the gap between approximate and accurate solutions for 15-task flows. This provides strong insights into the near-optimality of RO-III in practice although no real experiments are feasible in order to establish the ground truth for bigger flows and near optimality cannot be theoretically proved (most probably), as explained in Section 2.

The experiments above refer to uniformly distributed values of costs and selectivities. We repeat the experiments, when those values follow the beta distribution, which can describe selectivities, as explained in [26]. We have tested several parameters of that distribution, without big differences; here
we present the results when the two main beta distribution parameters are set to \(a = b = 0.5\). Table 3 presents the results of performance improvement of the RO-I, RO-II, RO-III and Swap heuristics normalized according to the cost of the initial randomly generated plan; the PCs are 40%. The last two columns of Table 3 are computed as follows over all 100 iterations: \(\text{AvgDiff} = \frac{1}{100} \sum \frac{\text{Swap} - \text{ROIII}}{\text{Swap}}\) and \(\text{MaxDiff} = \max\{\frac{\text{Swap} - \text{ROIII}}{\text{Swap}}\}\), and as such the closer the values to 1 the bigger the relative improvement of RO-III.

The main observation here is that for beta-distributed values, the performance of RO-III against Swap improves even more. In the case of flows that consist of 80 and 100 tasks, the RO-III results in 60% and 57% less SCM, which implies a 2.5x and 2.32x speed-up, respectively; this reduction is significantly higher than the one for uniformly distributed metadata. Interestingly, in a specific iteration, the maximum observed decrease is by 3 orders of magnitude. In general, especially for large flows, the performance improvements for beta-distributed values are higher for all techniques.

### Table 3: Normalized performance for data flows with 40% engine constraints

| n  | Initial | RO-I   | RO-II  | RO-III | Swap   | Avg Diff | Max Diff |
|----|---------|--------|--------|--------|--------|----------|----------|
| 20 | 1.0000  | 0.3539 | 0.3566 | 0.2841 | 0.4101 | 0.2636   | 0.8102   |
| 50 | 1.0000  | 0.2696 | 0.2679 | 0.1780 | 0.2761 | 0.3281   | 0.9802   |
| 80 | 1.0000  | 0.2181 | 0.2225 | 0.1420 | 0.2355 | 0.4069   | 0.9663   |
| 100| 1.0000  | 0.2149 | 0.2285 | 0.1478 | 0.2900 | 0.2980   | 0.9985   |

| n  | Initial | RO-I   | RO-II  | RO-III | Swap   | Avg Diff | Max Diff |
|----|---------|--------|--------|--------|--------|----------|----------|
| 20 | 1.0000  | 0.3509 | 0.4285 | 0.2837 | 0.4095 | 0.2756   | 0.9562   |
| 50 | 1.0000  | 0.3942 | 0.2287 | 0.1076 | 0.2310 | 0.4865   | 0.9898   |
| 80 | 1.0000  | 0.1356 | 0.1945 | 0.0553 | 0.1483 | 0.6041   | 0.9949   |
| 100| 1.0000  | 0.0944 | 0.1591 | 0.0538 | 0.1141 | 0.5699   | 0.9995   |

### 8.1.2. 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 6. To this end, we compare the parallel version of Swap, named as PSwap, against the parallel proposed rank ordering-based algorithms, denoted as PRO-I,PRO-II,PRO-III, respectively. We also compare against PGreedyII, which outperforms PGreedyI as shown in additional experiments in Appendix E. Initially, we assume that the merge cost \(mc\) is 0, but we relax this assumption later.

The comparisons are presented in Table 4 where it is shown that the
Table 4: Normalized performance for data flows with n=50,100 tasks.

| alg\PCs(%) | 20   | 40   | 60   | 80   |
|-----------|------|------|------|------|
| Initial   | 1.0000 | 1.0000 | 1.0000 | 1.0000 |
| PSwap     | 0.1759 | 0.2723 | 0.3293 | 0.4987 |
| PSwapⅠ    | 0.1804 | 0.2812 | 0.3448 | 0.5177 |
| PGreedyII | 0.1052 | 0.1839 | 0.2842 | 0.4413 |
| PGreedyIIⅠ| 0.1057 | 0.1865 | 0.2921 | 0.4552 |
| PRO-I     | 0.1340 | 0.2277 | 0.2949 | 0.4534 |
| PRO-IⅠ    | 0.1363 | 0.2321 | 0.3011 | 0.4629 |
| PRO-II    | 0.1171 | 0.2418 | 0.4455 | 0.5335 |
| PRO-IIⅠ   | 0.1188 | 0.2497 | 0.4686 | 0.5579 |
| PRO-III   | 0.0989 | 0.1600 | 0.2156 | 0.4012 |
| PRO-IIIⅠ  | 0.0990 | 0.1605 | 0.2166 | 0.4062 |

| alg\PCs(%) | 20   | 40   | 60   | 80   |
|-----------|------|------|------|------|
| Initial   | 1.0000 | 1.0000 | 1.0000 | 1.0000 |
| PSwap     | 0.0855 | 0.1428 | 0.2087 | 0.3440 |
| PSwapⅠ    | 0.0886 | 0.1488 | 0.2197 | 0.3580 |
| PGreedyII | 0.0485 | 0.0765 | 0.1274 | 0.2635 |
| PGreedyIIⅠ| 0.0485 | 0.0769 | 0.1299 | 0.2719 |
| PRO-I     | 0.0793 | 0.1264 | 0.2013 | 0.2994 |
| PRO-IⅠ    | 0.0820 | 0.1302 | 0.2072 | 0.3009 |
| PRO-II    | 0.0605 | 0.4507 | 0.2522 | 0.4783 |
| PRO-IIⅠ   | 0.0618 | 0.4911 | 0.2671 | 0.4728 |
| PRO-III   | 0.0465 | 0.0681 | 0.1058 | 0.2183 |
| PRO-IIIⅠ  | 0.0465 | 0.0681 | 0.1063 | 0.2204 |

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; for the moment we do not focus on those table rows. For linear flows, when n=50 and PCs=40%, RO-III results in decrease of the SCM of Swap by 32% (see Table 3). In a parallel setting, the decrease in SCM comparing PSwap and PRO-III reaches 41%. Also, for n=100, the performance improvements reach 52% (from 29%, see Table 3). The relative improvements are similar for PCs=60% and slightly less for PCs = 20% and PCs= 80%.

A question arises as to how often parallelization leads to benefits. Analyzing the individual runs, we have observed that the number of such occurrences is less than 10%, if we count only improvements higher than 2%.
Nevertheless, the magnitude of the improvements is strongly correlated with the number of PCs. For less constraints settings (PCs = 20%), for both $n=50$ and $n=100$, we have observed speed-ups of an order of magnitude. When PCs=40%, the maximum observed speed-up drops to 4 and 3 times, respectively. For even more PCs, this speed-up does not exceed 12.7%. A final note is that $PGreedyI$ is the best performing parallel heuristic from those not fully proposed in this work. The main conclusion up to here is that further refining the linear orderings with our proposed light-weight post-processing step can yield tangible performance improvements, and our proposals lead to further advancements in the current state-of-the-art in linear flow optimization.

Next, we repeat the experiments with non-zero merge cost, and the results verify that its impact is negligible (see Table 4). After real experiments with the PDI tool, we set $mc = 10$, that is an order of magnitude higher than the less expensive tasks and an order of magnitude lower than the most expensive ones. Overall, on average, our best performing solution, namely $PRO-III$ continues to have average performance improvements against $Swap$ of an order of magnitude.

8.1.3. Performance of MIMO flows

This set of experiments considers the evaluation of the methodology that is analyzed in Section 7 for MIMO data flow optimization. We consider two cases of butterfly flows (see Figure 9(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 percentage of PCs is 40%.

Figure 11 presents the average performance improvements of the $PRO-III$
and Swap algorithms over the non-optimized initial data flow. In the case where the linear segments are very small (10 tasks) the improvements are small as well. When the linear segment size increases to 20, PRO-III has 34% better performance improvement than Swap, and 74% lower execution cost compared to the non-optimized case. The performance improvements are commensurate with those in Table 3 which supports our claim that our proposals for SISO flows can be transferred to MIMO settings as well.

8.2. Time Overhead

In this section, we conduct a thorough evaluation of the time overhead of the accurate optimization algorithms. The purpose of this set of experiments is to show that the application of the exhaustive algorithms, and more specifically of TopSort, is limited only to small or very constrained flows.
Figure 12 (top-left) presents the average execution time of the DP algorithm compared to the TopSort solution for 50% precedence constraints. More specifically, this figure depicts the time overhead for executing data flows with \( n = 15, \ldots, 20 \) flow tasks. The main conclusions that can be drawn from this figure is that DP algorithm is not a practical optimization solution even for small flows that consist of 19 flow activities; the execution of a flow with 20 tasks requires over 3 days using our test machine. Even if the TopSort algorithm runs at least 50 times faster than DP, the execution of TopSort follows a similar pattern with DP.

Figure 12 (top-right) shows the average execution time of TopSort for flows with \( n = 10, \ldots, 70 \) having 98% precedence constraints, which implies that the number of the possible re-orderings is quite restricted. TopSort does not scale well, but can run in acceptable time even for medium-sized flows of 60 tasks. Additionally, Figure 12 (bottom-left) depicts that TopSort cannot scale for arbitrary precedence constraints even for flows with 15 and 20 flow activities. For example, the execution time of a data flow with 20 tasks and 50% precedence constraints is 2 orders of magnitude higher than the execution time of a data flow with 15 tasks. Finally, in the bottom-right part of Figure 12, the time overhead of Backtracking compared to TopSort is presented. The main observation of this figure, where the precedence constraints range is \( PCs = 90\%, \ldots, 98\% \), is that Backtracking can be up to 62 times slower than TopSort.

Overall, we can conclude that TopSort, on the one hand scales better than the other techniques and is applicable in specific cases where the other two approaches are not, but, on the other, it is not able to scale in general. We do not present the overhead of the approximate solutions, because it is negligible.

9. Related Work

The existing approaches of flow optimization can be classified in the following main categories, which are subsequently presented in turn:

- **Optimization of the structure of data flows**: this category targets the methodologies that optimize the flow execution plan through changes in the structure of the flow graph including task re-ordering.

- **Optimization of the resource allocation and scheduling aspects of data flows**: the proposals in this category deal with issues such as the alloca-
tion of computational resources and specific execution engines to each part of the flow along with time scheduling details, without affecting the workflow structure.

- Application-dependent solutions: this category contains optimization techniques that are specific to certain settings; interestingly, some of these techniques leverage database technologies.

Optimization of the structure of data flows. An aspect of this category that is particularly relevant to our work considers flow optimization inspired by query processing techniques. In [27], an optimization algorithm for query plans with dependency constraints between algebraic operators is presented. The adaptation of this algorithm in our SISO problem setting that does not consider only algebraic operators is reduced to the existing optimization algorithms we have presented in previous sections, and more specifically to GreedyI and Partition. In [28], ad-hoc query optimization methodologies are employed in order to perform structure reformations, such as reordering 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 [10]. Specifically, the authors consider ETL execution plans as states and use transitions, such as swap, merge, split, distribute and so on, to generate new states in order to navigate through the state space, which corresponds to the execution plan alternatives; they also present optimization algorithms for reducing ETL workflow execution cost albeit with exponential complexity. In our work, where we consider only task re-orderings, the proposal in [10] corresponds to the Swap algorithm, which we have presented and evaluated.

Another interesting approach to flow optimization is presented in [6], 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 reordering algorithms per se. In [29], they introduce a suite of quality metrics (QoX) without going into flow optimization algorithm details.

In addition, there is a significant portion of proposals on flow optimization that proceed to flow structure optimizations but do not perform task reordering, as we do. For example, an interesting proposal that aims to combine the control and the data flow view of workflows has appeared in [30]. That work presents approaches that merge tasks related to data management
to decrease the number of invocations to the underlying databases without changing the relative order of the tasks. In [31], a data oriented method for workflow optimization is proposed in order to minimize execution cost. This method is based on the fact that data may be shared across several functions, and, as such, workflow performance stands to benefit from optimizations in the form of incorporating a shared database to handle common data-oriented tasks. Another workflow optimization method that affects the workflow structure with a view to improving the efficiency of the workflow is presented in [32]. This method is inspired by the current limitations of business information processes. In particular, a task redesigning method is presented, which is based on the consolidation of the tasks to reduce the overall execution time. Quality of Service requirements (QoS), such as precedence of information flows and technology support costs are taken into account. In [33], a methodology to choose the optimal physical implementation of each task and decide whether to introduce special sorting tasks is presented, when there are several implementation alternatives. This work does not consider the execution order of the flow activities. Several optimizations in workflows are also discussed in [34], but the techniques are limited to straightforward application of query optimization techniques, such as join reordering and pushing down selections.

Optimization of the resource allocation and scheduling aspects of data flows. The main motivation of the proposals in this category stem from the need for more efficient resource management, given that resource management is deemed as a key performance factor. Contrary to our work, they assume an execution setting with multiple execution engines and do not deal with optimization of the flow task ordering. For example, in [35, 36], they introduce resource allocation algorithms and heuristic techniques that have the ability to take into account constraints, such as cost optimization, user-specified deadline and workflow partitioning according to assigned deadlines [36]. [2] discusses methodologies about how to execute and dispatch task activities in parallel computers.

Another family of optimization proposals deals with task scheduling methods, considering aspects such as semantic expression of workflow tasks, dynamic selection of services among many candidates and latency minimization [37, 38, 39, 40]. Also, there are scheduling methods which are exclusively related with grid workflow optimization (e.g., [37, 38, 39]), or linear workflow optimization, such as [40], which discusses optimal time schedules given a fixed allocation of activities to engines. Also, a set of optimization algo-
rithms based on deadline and time constraints was analyzed for scheduling flows in [41, 42]. Another proposal of flow optimization is presented in [43] based on soft deadline rescheduling in order to deal with the problem of fault tolerance in flow executions. In [44], an optimization methodology for minimizing the performance fluctuations that might occur by the resource diversity, which also considers deadlines, is proposed. Additionally, there is a set of optimization methodologies based on multi-objective optimization. For example, an auction-based scheduling methodology for multi-objective flow optimization is presented in [45], while [46, 47] propose optimization methodologies for multi-engine environments meeting multiple objectives, such as fault-tolerance and performance. The implementation of some of the presented optimization methods mentioned above is carried out with the help of algorithms that take into consideration certain quality of service requirements (QoS). In this case, users are responsible to set constraints, such as reliability, time, security, cost and fidelity, which are the principle parameters of workflow task scheduling. In this work, we do not consider resource allocation and scheduling issues, which are orthogonal to task ordering.

Application-dependent solutions. An important part of workflow optimization research was originated by optimization methods that have been created for a specific applications and as such, they are application dependent. An example of application dependent workflow optimization is discussed in [48], which deals with the creation and process of technical documents by a document workflow management system; in this work, the parallelism opportunities presented by the document structure are exploited to optimize workflows. Another example is [49], where a process execution management framework is proposed in order to optimize business objectives of processes in a dynamic business environment. Also, there are workflow optimization methodologies applied in other scientific fields. A representative example is [50], where the optimization algorithms are used for the development of molecular models and they are applied to a simulation tool. Analogous examples that achieve workflow optimization only under certain circumstances are presented in [51, 52, 53, 54]. However, these optimization methods cannot be adapted to a more general case.

10. 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, as, nowadays, the optimization of the complex data flows is left to the flow designers and is a manual procedure. Firstly, as the query optimization techniques are not applicable to data flow optimization because of the precedence constraints and the existing proposals for optimal solutions cannot scale, there is significant need to propose new flow optimization methodologies. We show that the state-of-the-art optimization algorithms can have 74% higher execution cost than the optimal solution even for the simplest type of single-input single-output (SISO) flows with a small number of tasks. So, to fill the gap of near-optimal optimization techniques, we propose a set of approximate algorithms that can exhibit 40% performance improvements than the best existing heuristic. We also introduce a post-process optimization phase for parallel execution of the flow tasks in order to improve even more the performance of a data flow, and we show that we can extend these solutions to more complex data flow scenarios that deal with arbitrary number of sources and sinks. This work aims to provide the basis for more holistic flow optimization algorithms, which do not only consider more complex flows, but also combine task ordering with aspects, such as task implementation and scheduling.

11. Acknowledgments

This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program “Education and Lifelong Learning” of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

References

[1] S. Chaudhuri, U. Dayal, V. Narasayya, An overview of business intelligence technology, Commun. ACM 54 (2011) 88–98.

[2] E. S. Ogasawara, D. de Oliveira, P. Valduriez, J. Dias, F. Porto, M. Mattoso, An algebraic approach for data-centric scientific workflows., PVLDB 4 (2011) 1328–1339.
[3] A. Simitsis, K. Wilkinson, M. Castellanos, U. Dayal, Optimizing analytic data flows for multiple execution engines, in: Proc. of the 2012 ACM SIGMOD Int. Conf. on Management of Data, 2012, pp. 829–840.

[4] A. Rheinländer, A. Heise, F. Hueske, U. Leser, F. Naumann, Sofa: An extensible logical optimizer for udf-heavy dataflows., CoRR abs/1311.6335 (2013).

[5] C. Olston, S. Chopra, U. Srivastava, Generating example data for dataflow programs, in: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, SIGMOD ’09, 2009, pp. 245–256.

[6] F. Hueske, M. Peters, M. Sax, A. Rheinländer, R. Bergmann, A. Krettek, K. Tzoumas, Opening the black boxes in data flow optimization, PVLDB 5 (2012) 1256–1267.

[7] R. Halasipuram, P. M. Deshpande, S. Padmanabhan, Determining essential statistics for cost based optimization of an etl workflow, in: EDBT, 2014, pp. 307–318.

[8] S. Chaudhuri, An overview of query optimization in relational systems., in: Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, ACM Press, 1998, pp. 34–43.

[9] Y. E. Ioannidis, Query optimization., ACM Comput. Surv. 28 (1996) 121–123.

[10] A. Simitsis, P. Vassiliadis, T. K. Sellis, State-space optimization of etl workflows, IEEE Trans. Knowl. Data Eng. 17 (2005) 1404–1419.

[11] R. Yerneni, C. Li, J. D. Ullman, H. Garcia-Molina, Optimizing large join queries in mediation systems, in: ICDT, 1999, pp. 348–364.

[12] Pentaho - data integration (kettle), http://kettle.pentaho.com, 2014.

[13] G. Kougka, A. Gounaris, Optimization of data-intensive flows: Is it needed? is it solved?, in: DOLAP, 2014, p. to appear.
[14] D. Zinn, S. Bowers, T. McPhillips, B. Ludäscher, Scientific workflow design with data assembly lines, in: Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science, WORKS ’09, ACM, 2009, pp. 14:1–14:10.

[15] J. Burge, K. Munagala, U. Srivastava, Ordering Pipelined Query Operators with Precedence Constraints, Technical Report 2005-40, Stanford InfoLab, 2005.

[16] U. Srivastava, K. Munagala, J. Widom, R. Motwani, Query optimization over web services, in: Proc. of the 32nd Int. Conference on Very large data bases VLDB, 2006, pp. 355–366.

[17] K. Agrawal, A. Benoit, F. Dufossé, Y. Robert, Mapping filtering streaming applications, Algorithmica 62 (2012) 258–308.

[18] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, T. G. Price, Access path selection in a relational database management system., in: Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, 1979, pp. 23–34.

[19] Y. L. Varol, D. Rotem, An algorithm to generate all topological sorting arrangements., The Computer Journal 24 (1981) 83–84.

[20] G. Koukga, A. Gounaris, Declarative expression and optimization of data-intensive flows, in: DaWaK, 2013, pp. 13–25.

[21] N. Kumar, P. S. Kumar, An efficient heuristic for logical optimization of etl workflows., in: BIRTE, 2010, pp. 68–83.

[22] T. Ibaraki, T. Kameda, On the optimal nesting order for computing n-relational joins, ACM Trans. Database Syst. 9 (1984) 482–502.

[23] R. Krishnamurthy, H. Boral, C. Zaniolo, Optimization of nonrecursive queries, in: VLDB, 1986, pp. 128–137.

[24] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, A. P. Barros, Workflow patterns, Distrib. Parallel Databases 14 (2003) 5–51.
[25] P. Vassiliadis, A. Karagiannis, V. Tziovara, A. Simitsis, Towards a benchmark for ETL workflows, in: Proceedings of the Fifth International Workshop on Quality in Databases, QDB 2007, at the VLDB 2007 conference, Vienna, Austria, September 23, 2007, 2007, pp. 49–60.

[26] B. Babcock, S. Chaudhuri, Towards a robust query optimizer: A principled and practical approach., in: SIGMOD Conference, 2005, pp. 119–130.

[27] D. Florescu, A. Levy, I. Manolescu, D. Suciu, Query optimization in the presence of limited access patterns, in: Proceedings of the 1999 ACM SIGMOD international conference on Management of data, SIGMOD ’99, ACM, 1999, pp. 311–322.

[28] G. Koungka, A. Gounaris, On optimizing workflows using query processing techniques, in: SSDBM, 2012, pp. 601–606.

[29] U. Dayal, M. Castellanos, A. Simitsis, K. Wilkinson, Data integration flows for business intelligence, in: Proc. of the 12th Int. Conf. on Extending Database Technology: Advances in Database Technology, EDBT, ACM, 2009, pp. 1–11.

[30] M. Vrhovnik, H. Schwarz, O. Suhre, B. Mitschang, V. Markl, A. Maier, T. Kraft, An approach to optimize data processing in business processes, in: VLDB, 2007, pp. 615–626.

[31] R. Minglun, Z. Weidong, Y. Shanlin, Data oriented analysis of workflow optimization, in: Proc. of the 3rd World Congress on Intelligent Control and Automation, 2000 - Volume 4, IEEE Computer Society, 2000, pp. 2564 – 2566.

[32] R. Dewan, A. Seidmann, Z. Walter, Workflow optimization through task redesign in business information processes, in: HICSS, IEEE Computer Society, 1998, pp. 240–252.

[33] V. Tziovara, P. Vassiliadis, A. Simitsis, Deciding the physical implementation of ETL workflows, in: Proc. of the ACM 10th Int. Workshop on Data warehousing and OLAP DOLAP, 2007, pp. 49–56.

[34] M. Böhm, Cost-based optimization of integration flows, Ph.D. thesis, 2011.
[35] R. Duan, R. Prodan, T. Fahringer, Performance and cost optimization for multiple large-scale grid workflow applications, in: Proceedings of the 2007 ACM/IEEE conference on Supercomputing, 2007, pp. 1–12.

[36] Z. Xiao, H. Chang, Y. Yi, Optimization of workflow resources allocation with cost constraint, in: Proc. of the 10th Int. Conf. on Computer supported cooperative work in design, Springer-Verlag, 2007, pp. 647–656.

[37] E. Schikuta, H. Wanek, I. Ul Haq, Grid workflow optimization regarding dynamically changing resources and conditions, Concurr. Comput.: Pract. Exper. 20 (2008) 1837–1849.

[38] W. N. Chen, J. Zhang, An ant colony optimization approach to a grid workflow scheduling problem with various qos requirements, Ieee Transactions on Systems Man and Cybernetics Part C Applications and Reviews 39 (2009) 29–43.

[39] V. S. Kumar, P. Sadayappan, G. Mehta, K. Vahi, E. Deelman, V. Ratnakar, J. Kim, Y. Gil, M. Hall, T. Kurc, J. Saltz, An integrated framework for performance-based optimization of scientific workflows, in: HPDC, ACM, 2009, pp. 177–186.

[40] K. Agrawal, A. Benoit, L. Magnan, Y. Robert, Scheduling algorithms for linear workflow optimization, in: IPDPS’2010, the 24th IEEE International Parallel and Distributed Processing Symposium, IEEE Computer Society Press, 2010.

[41] S. Abrishami, M. Naghibzadeh, D. H. J. Epema, Cost-driven scheduling of grid workflows using partial critical paths, IEEE Trans. Parallel Distrib. Syst. 23 (2012) 1400–1414.

[42] S. Abrishami, M. Naghibzadeh, D. H. Epema, Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds, Future Generation Computer Systems 29 (2013) 158 – 169.

[43] K. Plankensteiner, R. Prodan, Meeting soft deadlines in scientific workflows using resubmission impact, Parallel and Distributed Systems, IEEE Transactions on 23 (2012) 890–901.
[44] R. N. Calheiros, R. Buyya, Meeting deadlines of scientific workflows in public clouds with tasks replication, IEEE Transactions on Parallel and Distributed Systems 99 (2013) 1.

[45] H. Fard, R. Prodan, T. Fahringer, A truthful dynamic workflow scheduling mechanism for commercial multicloud environments, Parallel and Distributed Systems, IEEE Transactions on 24 (2013) 1203–1212.

[46] A. Simitsis, K. Wilkinson, M. Castellanos, U. Dayal, Optimizing analytic data flows for multiple execution engines, in: SIGMOD Conference, 2012, pp. 829–840.

[47] A. Simitsis, K. Wilkinson, U. Dayal, M. Hsu, Hfms: Managing the lifecycle and complexity of hybrid analytic data flows, in: ICDE, 2013.

[48] H. Dortmans, L. Somers, Document workflow optimization, in: Proceedings of the 14th European Simulation Symposium, 2002.

[49] M. J. Buco, R. N. Chang, L. Z. Luan, E. So, C. Tang, C. Ward, Pem: A framework enabling continual optimization of workflow process executions based upon business value metrics, in: Proc. of the IEEE International Conference on Services Computing, 2005, pp. 33–40.

[50] M. Hülsmann, T. Köllemann, J. Vrabec, D. Reith, Grow: A gradient-based optimization workflow for the automated development of molecular models, Computer Physics Communications 181 (2010) 499–513.

[51] B. I. Reiner, E. L. Siegel, J. A. Carrino, Workflow optimization: Current trends and future directions, J. Digital Imaging 15 (2002) 141–152.

[52] M. Faschinger, C. R. Sastry, A. H. Patel, N. C. Tas, An rfid and wireless sensor network-based implementation of workflow optimization, WoW-MoM (2007) 1–8.

[53] L. Huang, A. Akram, R. Allan, D. W. Walker, O. F. Rana, Y. Huang, A workflow portal supporting multi-language interoperation and optimization: Research articles, Concurr. Comput. : Pract. Exper. 19 (2007) 1583–1595.

[54] B. C. Pearce, D. R. Langley, J. Kang, H. Huang, A. Kulkarni, E-novo: an automated workflow for efficient structure-based lead optimization., Journal of Chemical Information and Modeling 49 (2009) 1797–1809.
Algorithm 5 Dynamic Programming

Require: A set of n tasks, T = \{t_1, ..., t_n\}
A directed acyclic graph PC with precedence constraints

Ensure: A directed acyclic graph P representing the optimal plan

- Initialize PartialPlan, Costs and Sel of size \(2^n - 1\)

1: for all \(i \in \{2, ..., n\}\) do
2:   PartialPlan[\(2^{i-1}\)] = t_i; Costs[\(2^{i-1}\)] = \(c_i\); Sel[\(2^{i-1}\)] = \(sel_i\);
3: end for
4: for all \(s \in \{2, ..., n\}\) do
5:   \(R \leftarrow Subsets(T, s)\) \{X is a set with all subsets of T of size s\}
6:   \(r\) is a specific subset of size \(s\)
7:   tempBest \(\leftarrow \infty\)
8:   for each \(r \in R\) do
9:     for all \(i \in \{1, ..., r.length()\}\) do
10:    tempSet \(\leftarrow r - r(i)\)
11:    pos1 \(\leftarrow findIndex(tempSet)\)
12:    pos2 \(\leftarrow findIndex(r(i))\)
13:    if sp(i) has all predecessors in tempSet then
14:       TempPlan \(\leftarrow tempSet, r(i)\)
15:       costTempPlan \(\leftarrow Costs[pos1] + Sel[pos1]Costs[pos2]\)
16:       if costTempPlan < tempBest then
17:          tempBest \(\leftarrow costTempPlan\)
18:          \(k \leftarrow pos1 + pos2\)
19:          update(PartialPlan[k], Costs[k], Sel[k])
20:     end if
21:   end for
22: end for
23: end for
24: P \(\leftarrow PartialPlan[2^n - 1]\)

Appendix A. Extra material about the DP algorithm

In order to implement the algorithm, we use three vectors of size \(2^n - 1\), namely PartialPlan, Costs and Sel. According to the algorithm implementation, the \(i\)-th cell corresponds to the combination of tasks for which the bit is 1 in its binary representation. For example, if \(i = 13\), then the binary representation of this position is (1101)\(_2\). Specifically, this means that partialPlan[13] corresponds to the optimal ordering of the 1\(^{st}\), 2\(^{nd}\) and 4\(^{th}\)
tasks. The Costs and Sel vectors hold the aggregate cost and selectivity of the subplans, respectively. The last cell of PartialPlan and Costs contain the optimal plan and its total cost, respectively. A complete pseudocode is shown
in Algorithm 5. For the sake of simplicity of presentation, the algorithm is not fully optimized; e.g., in line 18, the update of vertices may occur only once after the final best plan is found.

We give an example of the algorithm with a flow with \( n = 5 \); the task metadata are shown in Figure A.13. The DP example is in Figure A.14. First of all, all the subsets \( R \) of \( T \) of length \( K = \{1, 2, ..., n\} \) are found. For single task subsets, such as \( \{t_1\}, \{t_2\}, ..., \{t_n\} \), DP estimates their position in the partialPlan matrix, e.g. \( \{2\} \) subset is positioned in partialPlan(\( 2^{(2^1 - 1)} \), 1).

For subsets with length greater than 1, e.g., the subset \( \{1, 3, 4\} \), we examine the case that each element of that subset is placed at the end of the subset. If the precedence constraints are violated, DP continues to the next placement. If the precedence constrains are not violated, the algorithm estimates the cost of the valid partial plan with that element positioned at the end of the subset, reusing the results of the orderings of smaller subsets. Similarly, the cost of all orderings in the subset is estimated and the algorithm finds the ordering of the subset with the minimum cost. The optimal partial plan, its cost and the product of task selectivities are stored in the corresponding position in the partialPlan and DPCs vertices, respectively. For example, the partial plan \( \{1, 3, 4, 5\} \) is stored in position \( 2^{1-1} + 2^{3-1} + 2^{4-1} + 2^{5-1} = 29 \) of the partialPlan matrix.

Correctness: If PartialPlan is of size \( n = 1 \), the optimal solution is trivial and is found by the algorithm during initialization in lines 1-3 of Algorithm 5. We assume that a PartialPlan of size \( n - 1 \) is optimal and we need to prove that PartialPlan of size \( n \) is also optimal. The sketch of the proof will be based on contradiction. Let us assume that the DP does not produce the optimal solution. Any linear solution of size \( n \) consists of a PartialPlan of size \( n - 1 \) followed by the \( n \)-th task; DP checks all the alternatives for the \( n \)-th task. So, there is a different optimal solution, where the PartialPlan of size \( n - 1 \) is different of DP’s PartialPlan of the same size. According to the SCM, the cost of the subplan of size \( n \) is computed as the sum of two components: the cost of subplan of size \( n - 1 \) and the cost of the \( n \)-th task times the selectivity of the first \( n - 1 \) tasks. The costs of the solutions of size \( n \), which end with the same task, differ only in the first component. According to our assumptions, the cost of DP’s PartialPlan of size \( n - 1 \) cannot be higher than any other subplan solution of size \( n - 1 \) by definition. Consequently, there is no other solution different from DP’s solution that can yield lower cost. This completes the proof.
Algorithm 6 TopSort

Require: A set of n tasks, T={t₁, ..., tₙ} with known costs and selectivities.
A directed acyclic graph PC with precedence constraints.

Ensure: An ordering of the tasks P representing the optimal plan.

1: G={t₁, t₂, ..., tₙ} \{G is initialized with a valid topological ordering ordering of PC.\}
2: i=1
3: minCost ← computeSCM(G)
4: while i < n \{n is the total number of tasks\} do
5:   k ← location(1,i)
6:   k1 ← k + 1
7:   if G(k1) task has prerequisite i then
8:     // Rotation stage
9:     Rotate the elements of G from positions i to k
10:    cost ← computeSCM(G)
11:    i← i+1
12:  else
13:    // Swapping stage
14:    Swap the k and k1 elements of G
15:    cost ← computeSCM(G)
16:    i ← 1
17:  end if
18:  if cost < minCost then
19:    P ← G
20:    minCost = cost
21:  end if
22: end while

Appendix B. Extra material about the TopSort algorithm

The algorithm’s pseudocode is presented in Algorithm 6. The algorithm exhaustively checks all the permutations that satisfy the precedence constraints, and as such, it always finds the optimal solution for linear flows. The computeSCM function needs to be constructed in a way that does not compute the cost of each ordering from scratch, which is too naive, but leverages the computations of the previous plans taking into account the local changes in the new plan. In Figure B.15, an example of finding the optimal plan of a flow using TopSort is presented. In this example, the running steps of topSort algorithm are depicted, given as input a valid flow execution plan.
Figure B.15: Example of TopSort algorithm.

(Initial plan order plan label) and assuming the metadata of Figure A.13. Each of the given plans describe a plan generated after either a rotation or a swap action. The optimal flow execution plan is the one labeled Final plan order.

Note that we can implement TopSort in a different way, where the tasks are checked from right to left. Although in [19] this flavour is claimed to be capable of yielding better performance, this has not been verified in our flows.

Appendix C. Extra material about the existing approximate algorithms

Here we present the pseudocode for the Swap, GreedyI and Partition algorithms (Algorithms 7, 8, 10, respectively). Figures C.16, C.17 and C.18.
Algorithm 7 Swap

Require: A set of n tasks, $T=\{t_1, \ldots, t_n\}$
A directed acyclic graph $PC$ with precedence constraints

Ensure: A directed acyclic graph $P$ representing the optimal plan

1: $P \leftarrow \text{randomValidPlan}(PC)$ \{Initialize $P$\}
2: $\text{swapping} \leftarrow \text{true}$
3: while ($\text{swapping} == \text{true}$) do
4:    $\text{swapping} \leftarrow \text{false}$
5:    for all tasks $t_i \in T$ do
6:        if $t_{i+1}$ has not as prerequisite $t_i$ then
7:            if $(\text{computeSCM}(t_i \rightarrow t_{i+1}) < \text{computeSCM}(t_{i+1} \rightarrow t_i))$ then
8:                swap $t_i$ and $t_{i+1}$ in $P$
9:            end if
10:        end if
11:    end for
12: end while

Figure C.16: Example of Swap algorithm.

present examples for the input in Figure A.13.
**Algorithm 8 GreedyI**

**Require:** A set of $n$ tasks, $T = \{t_1, \ldots, t_n\}$

A directed acyclic graph $PC$ with precedence constraints

**Ensure:** A directed acyclic graph $P$ representing the optimal plan

1. $P \leftarrow \emptyset$
2. $Cand \leftarrow \emptyset$ \{Cand holds the candidate tasks\}
3. $C \leftarrow \emptyset$ \{C holds the considered tasks already in $P$\}
4. updateCandidates ($Cand, PC, C, T$)
5. **while** list $Cand$ is not empty **do**
6. **for** all tasks $t_j$ in $Cand$ **do**
7. Find task $t_j$ with maximum cost where cost = \((1 - sel_j)/cost_j\)
8. **end for**
9. Add $t_j$ task to optimal plan $P$
10. $C \leftarrow C \cup S_j$
11. updateCandidates ($Cand, PC, C, T$)
12. **end while**

**Algorithm 9 Function updateCandidates**

1. updateCandidates ($Cand, PC, C, T$)
2. **for** all tasks $t_i$ in $T$ **do**
3. **if** task $t_i \notin C$ **then**
4. **if** task $t_i$ has no prerequisites **then**
5. Add task $t_i$ to list $Cand$
6. **else**
7. **if** all of the prerequisites $\in C$ **then**
8. Add task $t_i$ to list $Cand$
9. **end if**
10. **end if**
11. **end if**
12. **end for**

**Appendix D. Extra material about the rank ordering-based techniques**

In this section, an illustrative example of the rank ordering methodologies is presented. Figure D.19 depicts metadata details for a data flow with 10 tasks, which are used as input for the application of $RO-I$, $RO-II$ and $RO-III$ algorithms. Specifically, this figure shows the PC graph, the values of selectivity and cost, but also the rank values that corresponds to each
task of the flow. We should mention that the sink node of the data flow is disconnected from the flow in the precedence constraint graph, as it is assumed that all the flow tasks must precede this task, and we connect it after the optimization procedure is finished. The detailed examples of the rank ordering proposals are described in extend in the following.

In Figure D.20, we present the pre-processing phase of the RO-I, in order to transform the precedence constraint graph into tree-shaped graph. The graph of the figure shows the final result of the dependency constraint graph. Then, we apply the KBZ algorithm, which is depicted in D.21.

---

Figure C.17: Example of Greedy algorithm.

**Greedy I algorithm**

| candidateTask | partialPlan |
|---------------|-------------|
| 1             | 1           |

\( t_1 \) is the only task that can precede all the others

| candidateTask | partialPlan |
|---------------|-------------|
| 2 3           | 1 3         |

\( \text{cost}(t_2) = 10 < \text{cost}(t_3) = 15 \)

| candidateTask | partialPlan |
|---------------|-------------|
| 2             | 1 3 2       |

\( t_2 \) is the only candidate task

| candidateTask | partialPlan |
|---------------|-------------|
| 4             | 1 3 2 4     |

\( t_4 \) is the only candidate task

\( P = 1 3 2 4 5 \)

\( \text{SCM}(P) = 14.01 \)

Figure C.18: Example of Partition algorithm.

**Partition algorithm**

| partition | partialPlan |
|-----------|-------------|
| 1         | 1           |

\( t_1 \) is the only task that can precede all the other tasks

| partition | partialPlan |
|-----------|-------------|
| 2 3       | 1 2 3       |

\( \text{cost}(t_2) = 18 > \text{cost}(t_3) = 13 \)

| partition | partialPlan |
|-----------|-------------|
|           | 1 2 3 4     |

\( t_5 \) is the only candidate task

\( P = 1 3 2 4 5 \)

\( \text{SCM}(P) = 12.01 \)
Algorithm 10 Partition

Require: A set of n tasks, T={t₁, ..., tₙ}
A directed acyclic graph PC with precedence constraints

Ensure: A directed acyclic graph P representing the optimal plan

1: P ← ∅
2: Cand ← ∅ {Cand holds the candidate tasks}
3: C ← ∅ {C holds the considered tasks already in P}
4: updateCandidates (Cand, PC, C, P)
5: while (Cand != ∅) do
6:   Estimate all possible permutations of the tasks tᵢ ∈ Cand
7:   tempBestCost ← 0
8:   tempBestPlan ← ∅
9:   for each possible permutation perm do
10:      costPerm ← computeSCM(permCand)
11:      if (costPerm < tempBestCost) then
12:         tempBestCost ← costPerm
13:         tempBestPlan ← perm
14:     end if
15: end for
16: Append perm to P
17: C ← C ∪ Cand
18: updateCandidates (Cand, PC, C, T)
19: end while

Figure D.19: The precedence constraint graph (PC), cost, selectivity, rank values of a data flow with 10 activities and the total execution cost.
Figure D.20: The pre-processing phase of RO-I to ensure that there are not cycles in the PC graph.

In the following, the validity post-process phase of RO-I is analyzed in D.22 and ensures that the optimized execution flow plan does not violate the dependency constraints. Finally, as is shown in this figure, the cost of the optimized execution plan is 237.0844.

The Figure D.23 illustrates in detail the steps of the application of RO-II. 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. The cost of the optimized flow plan returned by RO-II methodology is 317.3132.

In Figure D.24, the result of the post-processing phase of algorithm RO-III is described. In this phase the optimized flow plan occurred by moving the flow task $t_7$ to a later stage. The optimized cost of the flow execution is 205.5607.
Figure D.21: The optimization phase of RO-I by applying the KBZ algorithm.
Figure D.22: The validity phase of RO-I that ensures that there are not precedence constraint violations in the optimized execution plan.

Figure D.23: An application example of RO-II with the metadata of Figure D.19

Appendix E. Extra material about the PGreedy algorithms
Figure D.24: The post-process phase of the RO-III algorithm taking as input the generated optimized execution plan of RO-II, as depicted in D.23.

PGreedyI algorithm is shown in Algorithm 11. In this methodology the computation of each task cost was considered by two flavors. The first one is similar with the cost metric in [16], where the cost of the task is defined as equal to $inp_i c_i$ in each step. In this case, we add the candidate task that minimizes the $inp_i c_i$ to the optimal partial plan. In the second flavor PGreedyII the cost metric becomes $(1 - sel_i) / (inp_i c_i)$. This metric takes into account the selectivity of the next service to be appended in the execution plan and not only the selectivity of the preceding services. In Figure E.25 an example of the PGreedyI algorithm application based on the second cost metric is analyzed, given the cost, selectivity values, but also the precedence constraints.

In Figure E.26 the evaluation results of the performance improvement of the PGreedy flavours are shown. In this experiment, we compare our proposal of PGreedy optimization algorithm with its rank-based flavour, denoted as PGreedyII, but also each of these flavours is compared with the Swap heuristic and the initial plan cost. The presented performance results of Figure E.26 are normalized by the cost of the initial randomly generated flow execution plan. In Figure E.26 (left), the PGreedy has up to 95% better performance improvement than the initial plan cost, whereas the execution cost of PGreedyII can be up to 97% lower than the initial one. In most of the iterations, PGreedyRank seems to be clear winner. In the worst case,
Algorithm 11 PGreedy

Require: A set of n tasks, $T = \{t_1, ..., t_n\}$
A directed acyclic graph $PC$ with precedence constraints

Ensure: A directed acyclic graph $P$ representing the optimal plan

1: Initialize an adjacency matrix $P$ of optimal plan as empty
2: Initialize a list $Cand$ of candidate tasks as empty
3: Initialize a list $C$ of considered tasks as empty
4: updateCandidates ($Cand, PC, C, P$)
5: while list $Cand$ is not empty do
6: for all tasks $t_j$ in $Cand$ do
7: $v_j \leftarrow$ optimal value using a linear programming technique, which determines the optimal cost of adding $t_j$ in optimal plan $P$
8: $Cut_j \leftarrow$ optimal cut for adding $t_j$ \{cut: set of tasks that are the immediate predecessors\}
9: end for
10: $t_{opt} \leftarrow$ task having the least $v_j$
11: $Cut_{opt} \leftarrow$ optimal cut for adding $t_{opt}$
12: Add $t_{opt}$ task to optimal plan $P$ while directed edges from the tasks in $Cut_{opt}$ to $t_{opt}$
13: $C \leftarrow C \cup T_{opt}$
14: updateCandidates ($Cand, PC, C, P$)
15: end while
16: computeCost($P, costs, selectivities$)

$PGreedyII$ improves the performance of the non-optimized plan by no less than 54% on average. Also, $Swap$ in the best case has up to 89% better performance improvement than the initial flow plan. For 80% precedence constraints, as Figure E.26 shows, the $PGreedyII$ algorithm outperforms the other algorithms in all the data flows scenarios, even if the performance improvement decreases on average because of the limited possible reorderings. Specifically, in the best case, which is a flow with 70 tasks, $PGreedyRank$ has 74% lower execution cost, while $Swap$ improves the initial execution cost by 58%.
Figure E.25: Example of PGreedy algorithm.

Figure E.26: Performance improvement for data flows with $n \in [10, 100]$ with 40% precedence constraints (left) and 80% precedence constraints (right).