Optimal task ordering in chain data flows: exploring the practicality of non-scalable solutions

Georgia Kougka and Anastasios Gounaris

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

Abstract. Modern data flows generalize traditional Extract-Transform-Load and data integration workflows in order to enable end-to-end data processing and analytics. The more complex they become, the more pressing the need for automated optimization solutions. Optimizing data flows comes in several forms, among which, optimal task ordering is one of the most challenging ones. We take a practical approach; motivated by real-world examples, such as those captured by the TPC-DI benchmark, we argue that exhaustive non-scalable solutions are indeed a valid choice for chain flows. Our contribution is that we thoroughly discuss the three main directions for exhaustive enumeration of task ordering alternatives, namely backtracking, dynamic programming and topological sorting, and we provide concrete evidence up to which size and level of flexibility of chain flows they can be applied.

1 Introduction

Data analysis in a highly dynamic environment becomes more and more critical in order to extract high-quality information from raw data and derive actionable information in a timely manner. To this end, we typically employ fully automated *data-centric flows* (or *data flows*) both for business intelligence [4, 10] and scientific purposes [13], which typically execute under demanding performance requirements, e.g., to complete in a few seconds. These flows generalize traditional Extract-Transform-Load (ETL) and data integration flows through the incorporation of data analytics [8, 17]. Meeting the demanding performance requirements, combined with the volatile nature of the environment and the data, gives rise to the need for efficient data flow optimization techniques.

Data flow optimization techniques cover a wide spectrum from deciding on the order of the constituent tasks to detailed low-level configuration of the underlying execution engine [12]. In this work, we focus on the former aspect, namely 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 *chain* (or *linear*) 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, discussed in [3, 9], are that the tasks need not belong to a set of well-defined algebra, such as the relational one, there exist arbitrary precedence constraints among operators, and flows can consist of dozens of tasks, whereas, typically, operators in query plans are fewer. 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 . 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 [16, 20, 8]. A common characteristic of these proposals is that they are too slow to find an exact solution in small flows [8], or they can find significantly suboptimal (approximate) solutions for bigger flows [16, 20].

In this work, we go beyond the state-of-the-art with regards to exact solutions in chain data flows, i.e., data flows where the tasks form a sequence. Chain data flows are a main building block in generic data flows. Optimization of chain flows is a big step towards optimization of more generic ones. Exact solutions cannot scale in general, but, partially inspired by real worlds-like data flows, such as those in the TPC-DI benchmark [14], we show that they can be applied in several cases in practice. The main contribution of this work is the proposal of two additional exact solutions that significantly improve upon the technique in [8] in terms of time overhead and the size of data flows they can handle.[1]

The remainder of this paper is structured as follows. In Section 2, we present the notation, the problem statement and the motivation from TPC-DI. Our exact solutions are explained and evaluated in Sections 3 and 4, respectively. We mention related work in Section 5, and we conclude in the next section, which discusses the issues in optimizing more generic flows than chain ones.

2 Preliminaries

In this paper, we deal with the problem of re-ordering the tasks of a chain data flow without violating existing precedence constraints between tasks, while the performance of the flow is maximized. The data flow is represented as a

[1] An abstract of these ideas, without considering TPC-DI, have appeared in [11] in less than a page.

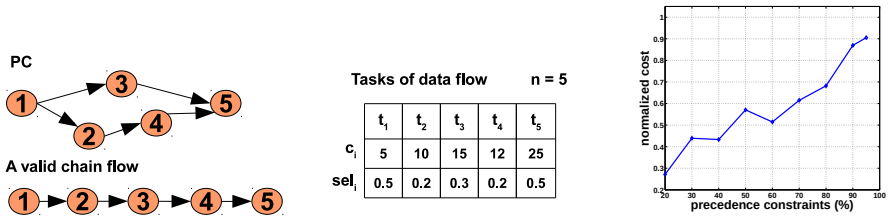

Fig. 1. An example of PC and G (left), task metadata (middle) and indicative performance improvements (right).

directed acyclic graph (DAG), denoted as $G = (T, E)$, where each task $t_i \in T$, $i = 1, \ldots, n$ 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 tasks that have no incoming edges are termed as *sources*, and those without outgoing edges as *sinks*.

The precedence constraints are captured by another directed acyclic graph $PC = (T, D)$, such that each ordered pair of vertices (t_i, t_j) that either belongs to D or the transitive closure of PC corresponds to the requirement that, in any valid G, there must exist a directed path from t_i to t_j. In other words, the PC graph corresponds to a higher-level, non-executable flow representation, where the exact task ordering is not defined; only a partial ordering is defined instead.

We focus on linear or chain data flows, where the flow contains only one sink and one source and each task in between has exactly one incoming and outgoing edge. Chain data flows can be regarded as sub-flows within generic data flows. For example, each path from a source to a sink in a generic flow forms a chain.

Further, we assume that each task t_i has a processing cost per input record (or tuple) c_i, and selectivity sel_i. The selectivity denotes the average number of returned tuples per input tuple. For filtering tasks, $sel_i < 1$; 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, $sel_i > 1$. Figure 1(left) shows an example of a chain data flow with 5 tasks, their precedence constraints, and one possible valid G that respects such constraints. In the middle, example task metadata are presented. As will be shown later (e.g., Figure 5), there are multiple other task orderings that respect the constraints as well, e.g., placing t_3 after t_4.

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 task ordering to form a chain G that minimizes the *sum cost metric (SCM)* per source tuple. SCM is defined as follows:

$$SCM(G) = \sum_{i=1}^{n} \Big(\prod_{j \in Pred(t_i)} sel_j \Big)\, c_i,$$

where $Pred(t_i)$ is the set of tasks that precede t_i in G; if the set is null, the product of selectivities is set to 1. In the example of Figure 1, $Pred(t_4) = \{t_1, t_2, t_3\}$. The optimal plan is denoted as P.

Figure 1(right) shows indicative performance improvements. We examine 100 randomly generated data flows consisting of $n = 15$ tasks with $c_i \in [1, 100]$, $sel_i \in (0, 2]$ and 20%-95% precedence constraints. A chain flow has 100% precedence constraints, when the transitive closure of its PC has $\frac{n(n-1)}{2}$ edges. The case of 100% precedence constraints is when there is a single valid ordering with no ordering alternatives. We use the percentage of precedence constraints as an efficient way to quantify the flexibility in reordering operators in a flow. The initial random ordering has normalized cost 1. As can be observed from Figure 1(right) , the improvements can be of a factor of 3 and more.

Note that the input set of tuples are processed by all the tasks of the chain data flow until they are filtered out, but typically, some of the input tuple attributes may not be required by every flow activity. According to [21], 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 [2], Burge et al 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 [2] it is discussed that *"it is unlikely that any polynomial time algorithm can approximate the optimal plan to within a factor of $O(n^\theta)$"*, where θ is some positive constant. Note that if we modify the optimization criterion, e.g., to optimize the bottleneck cost metric or the critical path renders the problem tractable [18, 1].

2.2 Chains in TPC-DI

TPC-DI [14] is the TPC standard for data integration flows. Although it focuses on extensions to ETLs rather than on both advanced ETLs and analytics, it is the closest standard to our scenarios. TPC-DI aims to model the data integration processes of a retail brokerage firm. More importantly, it has been implemented for an open-source data flow engine, namely Pentaho Kettle.[2] Figure 2 shows an example of part of the implementation, which is responsible for building one

[2] `http://www.essi.upc.edu/dtim/blog/post/tpc-di-etls-using-pdi-aka-kettle`

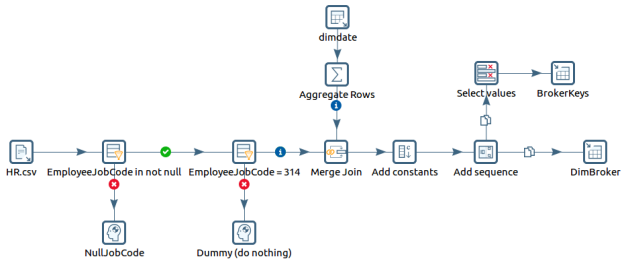

Fig. 2. A TPC-DI flow implemented in Kettle.

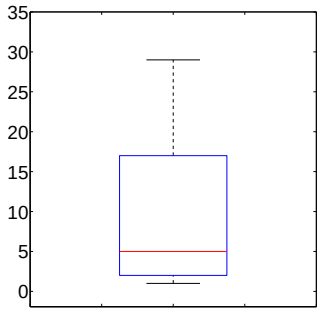

Fig. 3. Boxplot diagram for the length of chains in TPC-DI flows.

of the seven dimension tables in the underlying data warehouse defined by the standard.

Next, we proceed to a simple analysis of the flows referring to historical loading, as these are implemented in Kettle. First, we extract chain sub-flows according to the following procedure: for each sink, we take the longest path from any of the connected sources. The boxplot diagram in Figure 3 reveals significant information regarding the size of such chains. More specifically, most of the chains considered contain less than 20 tasks. Also, only 2 chains have size larger than 30 tasks; these chains are removed from the boxplot as outliers.

Analyzing the percentage of precedence constraints, it is found that there is a small degree of flexibility in re-ordering tasks. For example, in the largest chain in the boxplot, there are 97% precedence constraints. This pattern appears in all flows with size more than the median. For smaller chains, such as those contained in the flow in Figure 2, the constraints drop to 87%, which is still high. However, according to the evidence in Figure 1(right), significant improvements of 10-15% can still be achieved.

3 Accurate Algorithms for Linear Execution Plans

In this section, we present three accurate algorithms for reordering chain 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 not only for the type of flow chains appearing in TPC-DI, but also for much larger 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 very large size.

3.1 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 [8].

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.

3.2 Dynamic programming

This algorithm is extensively used as part of the System R-type of query optimization to produce (linear) join orderings [15]. 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 nth 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 $2^n - 1$.

Complexity: The time complexity is $O(n^2 2^n)$. 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 $O(2^n)O(n)O(n) = O(n^2 2^n)$. The space complexity is derived by the size of the

Algorithm 1 Dynamic Programming

Require: 1. A set of n tasks, T=$\{t_1, ..., t_n\}$. 2. 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$(2^{i-1}) = t_i$
3: Costs$(2^{i-1}) = c_i$
4: Sel$(2^{i-1}) = sel_i$;
5: **end for**
6: **for** all $s \in \{2, ..., n\}$ **do**
7: $R \leftarrow Subsets(T, s)$ {X is a set with all subsets of T of size s}
{r is a specific subset of size s}
8: $tempBest \leftarrow \infty$
9: **for** each $r \in R$ **do**
10: **for** all $i \in \{1, ..., r.length()\}$ **do**
11: $tempSet \leftarrow r - r(i)$
12: $pos1 \leftarrow findIndex(tempSet)$
13: $pos2 \leftarrow findIndex(r(i))$
14: **if** r(i) has all predecessors in *tempSet* **then**
15: $TempPlan \leftarrow tempSet, r(i)$
16: $costTempPlan \leftarrow Costs(pos1) + Sel(pos1)Costs(pos2)$
17: **if** $costTempPlan < tempBest$ **then**
18: $tempBest \leftarrow costTempPlan$
19: $k \leftarrow pos1 + pos2$
20: update(*PartialPlan(k)*, *Costs(k)*, *Sel(k)*)
21: **end if**
22: **end if**
23: **end for**
24: **end for**
25: **end for**
26: $P \leftarrow PartialPlan(2^n - 1)$

auxiliary data structures employed. We use three vectors of size $2^n - 1$ as explained in the implementation details, one of which stores elements of size $O(n)$. So the space complexity is $O(n2^n)$.

Implementation Issues: 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. Analogously, 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. 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 1. For the sake of simplicity of

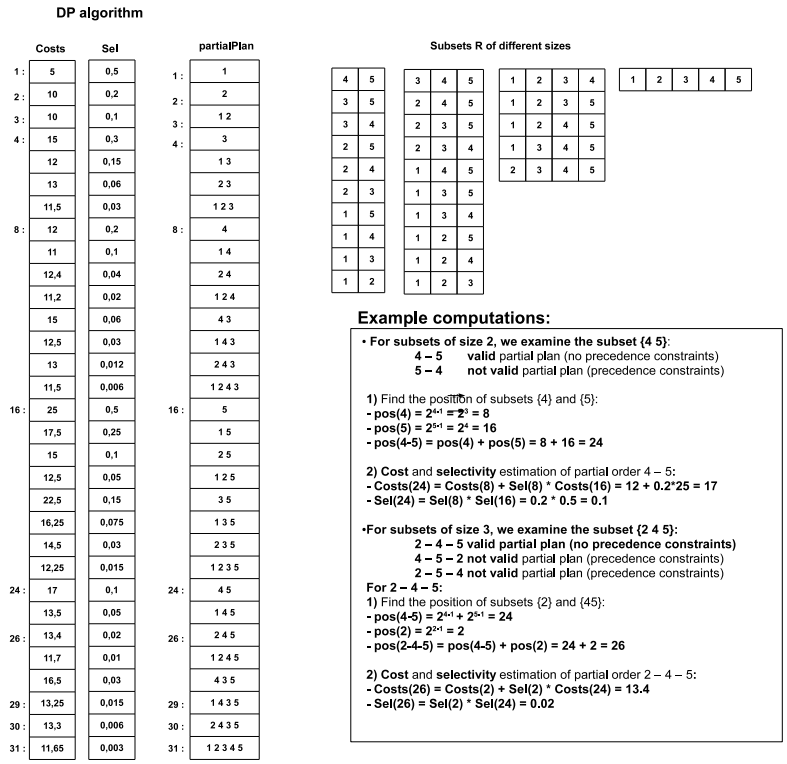

Fig. 4. Example of the DP algorithm.

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.

An example: We give an example of the algorithm with a flow with $n = 5$; the task metadata are shown in Figure 1. The *DP* example is in Figure 4. First of all, all the subsets R of T of length $s = \{1, 2, ..., n\}$ 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 $partialPlan(2^{2-1})$. 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 $Costs$ and Sel vertices, respectively.

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 1.

Algorithm 2 TopSort

Require: 1. A set of n tasks, T=$\{t_1, ..., t_n\}$. 2. A directed acyclic graph PC with precedence constraints.
Ensure: An ordering of the tasks P representing the optimal plan.

```
 1: P={t_1, t_2, ..., t_n} {P is initialized with a valid topological ordering of PC.}
 2: i=1
 3: minCost ← computeSCM(P)
 4: while i < n {n is the total number of tasks} do
 5:    k ← location of t_i in P
 6:    k1 ← k + 1
 7:    if P(k1) task has prerequisite t_i then
 8:       // Rotation stage
 9:       Rotate the elements of P from positions i to k
10:       cost ← computeSCM(P)
11:       i← i+1
12:    else
13:       // Swapping stage
14:       Swap the k and k1 elements of P
15:       cost ← computeSCM(P)
16:       i ← 1
17:    end if
18:    if cost < minCost then
19:       bestP ← P
20:       minCost = cost
21:    end if
22: end while
23: P ← bestP
```

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.

3.3 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;

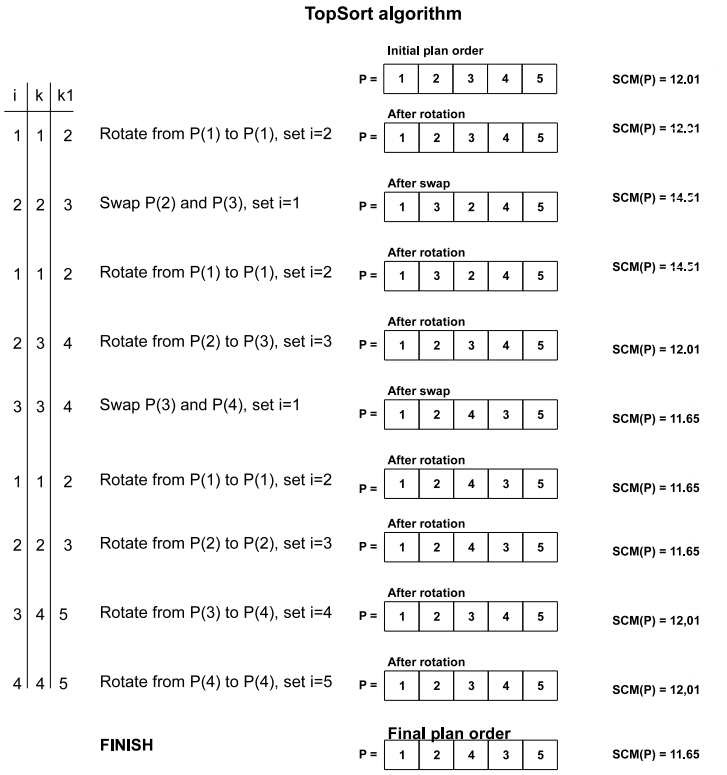

Fig. 5. Example of the TopSort algorithm.

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 two main operations, namely, 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 task 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 each generated plan, we estimate the total execution cost. A pseudocode is pesented in Algorithm 2.

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.

Implementation Issues: The algorithm exhaustively checks all the permutations that satisfy the precedence constraints, and as such, it always finds the optimal solution for linear flows. No specific data structures are required. As shown in Algorithm 2, we employ a *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. 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.

Example: In Figure 5, 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 (*Initial plan order* plan label) and assuming the metadata of Figure 1. Each of the given plans describe a plan generated after either a rotation or a swap action.

4 Evaluation of the Time Overhead

In this section, we conduct a thorough evaluation of the time overhead of the accurate optimization algorithms. We use a machine with an Intel Core i5 660 CPU and 6 GB of RAM. We construct synthetic flows so that we can evaluate the algorithms in a wide range of parameter combinations. More specifically, we produce the optimal ordering of tasks to form a chain flow after we have (i) created random PC graphs with a configurable ratio of precedence constraints, and (ii) chosen task metadata randomly. The time overhead does not actually depend on the task metadata.

Backtracking cannot scale in the percentage of PCs and *DP* cannot scale in the number of tasks. So, we compare them against *TopSort* separately.

Figure 6(top-left) presents the average execution time of the *DP* algorithm compared to the *TopSort* solution for 50% precedence constraints, and $n = 15, ..., 20$ flow tasks. For this ratio of PCs, *Backtracking* is practically inapplicable. 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 for this number of constraints; the execution of a flow with 20 tasks requires over 3 days using our test machine. *TopSort* runs at least

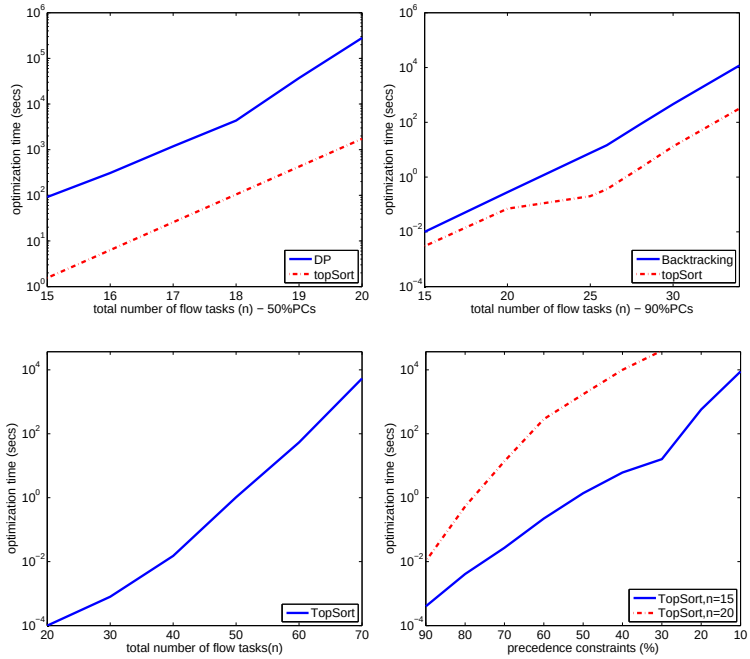

Fig. 6. Optimization overhead in several settings.

50 times faster than *DP*, but its execution follows a similar pattern to *DP*. However, this figure is indicative regarding the superiority of *TopSort* over *DP*. In the top-right part of Figure 6, the time overhead of *Backtracking* compared to *TopSort* is presented for $n = 15, \ldots, 34$ and 90% PCs. The main observation of this figure is that *Backtracking* is orders of magnitude slower than *TopSort* and cannot scale even for medium-size flows of more than 30 tasks.

We now turn our attention to *TopSort* for further investigation. Figure 6(bottom-left) shows the average execution time of *TopSort* for flows with $n = 10, ..., 70$ having 98% precedence constraints, which implies that the number of the possible re-orderings is quite restricted, as also observed in TPC-DI. *TopSort* does not scale well, but the important thing is that it can run in acceptable time, e.g., a minute, even in flows with 60 tasks. Additionally, Figure 6 (bottom-right) depicts that *TopSort* cannot scale for arbitrarily few precedence constraints even for flows with 15 and 20 flow activities. But, for these flow sizes, it can tolerate percentages of precedence constraints much lower than 90%.

5 Related Work

Further to the discussion of related work in the introduction, task ordering for data flows is an area that has been significantly influenced by query processing techniques. In [5], an optimization algorithm for query plans with dependency

constraints between algebraic operators is presented. In [20], a proposal for data integration that is also applicable to data flow task ordering is discussed. The corresponding techniques are approximate rather than exact solutions for chain flows, and they are either similar or inferior to those considered in [11]. ETL flows are analyzed in [16], where ETL execution plans are considered as states and transitions, such as swap, merge, split, distribute and so on, are employed to generate new states in order to navigate through the state space, which corresponds to the execution plan alternatives. As shown in [11], these techniques can deviate from the optimal solutions in chain dataflows by several factors.

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; an overview of the complete spectrum of data flow optimization techniques appears in [12]. Note that deciding the task ordering before execution in order to determine the flow structure is orthogonal to deciding the scheduling order of tasks at runtime, which is another well investigated area, e.g., see [6].

6 Conclusions

In this work, we deal with the problem of optimally ordering the constituent tasks of a chain 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. We are also motivated by the fact that real-world flows, such as those in TPC-DI, are not very flexible, in terms of the alternative valid task orderings. As such, carefully crafted exhaustive solutions become applicable. We propose two such solutions, a dynamic programming one and one that efficiently generates all topological orderings. We explain the technical details involved and we show that the latter approach is both dominant and practical under realistic assumptions.

Our work is a big step towards optimal task ordering in generic flows instead of simple chain flows, e.g., the complete flows assumed by the TPC-DI benchmark. For this goal to be met, three further issues need to be resolved: (i) to devise solutions that, using chain optimization as a building block, apply to complete data flows; in the way we defined chains hereby, these chains are overlapping and their isolated optimizations may not be compatible among overlapping chains and do not guarantee optimality when combined; (ii) to develop efficient ways to collect the required statistical metadata and detect precedence constraints; and (iii) to take into account other metrics than the minimization of the sum of the costs with a view to considering realistic issues, such as pipelined execution and parallel execution on multi-core engines, which better reflect the running time of a flow.

References

1. Kunal Agrawal, Anne Benoit, Fanny Dufossé, and Yves Robert. Mapping filtering streaming applications. *Algorithmica*, 62(1-2):258–308, 2012.

2. Jen Burge, Kamesh Munagala, and Utkarsh Srivastava. Ordering pipelined query operators with precedence constraints. Technical Report 2005-40, Stanford InfoLab, 2005.
3. Surajit Chaudhuri. An overview of query optimization in relational systems. In *PODS*, pages 34–43, 1998.
4. Surajit Chaudhuri, Umeshwar Dayal, and Vivek Narasayya. An overview of business intelligence technology. *Commun. ACM*, 54:88–98, 2011.
5. Daniela Florescu, Alon Levy, Ioana Manolescu, and Dan Suciu. Query optimization in the presence of limited access patterns. In *SIGMOD*, pages 311–322. ACM, 1999.
6. Xavier Grehant, Isabelle Demeure, and Sverre Jarp. A survey of task mapping on production grids. *ACM Comput. Surv.*, 45(3):37:1–37:25, July 2013.
7. Ramanujam Halasipuram, Prasad M. Deshpande, and Sriram Padmanabhan. Determining essential statistics for cost based optimization of an etl workflow. In *EDBT*, pages 307–318, 2014.
8. F. Hueske, M. Peters, M. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening the black boxes in data flow optimization. *PVLDB*, 5(11):1256–1267, 2012.
9. Yannis E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28(1):121–123, 1996.
10. Petar Jovanovic, Oscar Romero, and Alberto Abelló. A unified view of data-intensive flows in business intelligence systems: A survey. *T. Large-Scale Data- and Knowledge-Centered Systems*, 29:66–107, 2016.
11. Georgia Kougka and Anastasios Gounaris. Optimization of data-intensive flows: Is it needed? is it solved? In *DOLAP*, page to appear, 2014.
12. Georgia Kougka, Anastasios Gounaris, and Alkis Simitsis. The many faces of data-centric workflow optimization: A survey. *CoRR*, abs/1701.07723, 2017.
13. Eduardo S. Ogasawara, Daniel de Oliveira, Patrick Valduriez, Jonas Dias, Fabio Porto, and Marta Mattoso. An algebraic approach for data-centric scientific workflows. *PVLDB*, 4:1328–1339, 2011.
14. Meikel Poess, Tilmann Rabl, and Brian Caufield. TPC-DI: the first industry benchmark for data integration. *PVLDB*, 7(13):1367–1378, 2014.
15. Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.
16. A. Simitsis, P. Vassiliadis, and T. K. Sellis. State-space optimization of etl workflows. *IEEE Trans. Knowl. Data Eng.*, 17(10):1404–1419, 2005.
17. A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal. Optimizing analytic data flows for multiple execution engines. In *SIGMOD*, pages 829–840, 2012.
18. U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query optimization over web services. In *Proc. of the 32nd Int. Conference on Very large data bases VLDB*, pages 355–366, 2006.
19. Yaakov L. Varol and Doron Rotem. An algorithm to generate all topological sorting arrangements. *The Computer Journal*, 24(1):83–84, 1981.
20. Ramana Yerneni, Chen Li, Jeffrey D. Ullman, and Hector Garcia-Molina. Optimizing large join queries in mediation systems. In *ICDT*, pages 348–364, 1999.
21. Daniel Zinn, Shawn Bowers, Timothy McPhillips, and Bertram Ludäscher. Scientific workflow design with data assembly lines. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, WORKS '09, pages 14:1–14:10. ACM, 2009.