Optimization of Data-intensive Flows: Is it Needed? Is it Solved?

Georgia Kougka
Aristotle University of Thessaloniki
georkoug@csd.auth.gr

Anastasios Gounaris
Aristotle University of Thessaloniki
gounaria@csd.auth.gr

ABSTRACT

Modern data analysis is increasingly employing data-intensive flows for processing very large volumes of data. As the data flows become more and more complex and operate in a highly dynamic environment, we argue that we need to resort to automated cost-based optimization solutions rather than relying on efficient designs by human experts. We further demonstrate that the current state-of-the-art in flow optimizations needs to be extended and we propose a promising direction for optimizing flows at the logical level, and more specifically, for deciding the sequence of flow tasks.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

Keywords

data flow optimization; task reordering

1. INTRODUCTION

Nowadays, not only more and more data is produced, but there is also an increasing need for end-to-end processing of this data. End-to-end processing includes tasks, such as cleaning, extraction, integration and analytics, and as such, gives rise to data-intensive flows that go beyond traditional ETL (Extract-Transform-Loading) flows; the latter are restricted to simpler transformation task sequences and purpose, namely to populate a data warehouse. Data-intensive flows are encountered in both business intelligence [4] and scientific [13] settings.

Currently, data flows are typically designed manually, although commercial tools may provide some simple, static, cost-oblivious rule-based optimizations [6, 7]. Interestingly, there is an increasingly large portion of flow designers that are not IT experts [1], which raises doubts about the optimality of such manual designs. In addition, the optimality of a data flow depends on statistics, such as task costs and selectivities, which means that an optimal flow execution plan for a specific dataset may become sub-optimal when applied to another dataset with different statistics, or even for the same dataset if its characteristics evolve, as typically occurs in streaming real-time analytics. Because of all these factors, rather than relying on the skills of the designer, we need to resort to automated cost-based optimization solutions.

DOLAP'14, November 7, 2014, Shanghai, China.

ACM 978-1-4503-0999-8/14/11 ...$15.00.
http://dx.doi.org/10.1145/2666158.2666174 .

Flow execution can be defined both at *logical* and the *physical* level. At the logical level, the partial order of the tasks is typically represented as a directed acyclic graph (DAG), which describes the flow of the data from the source tasks to the sink ones and the exact sequence of tasks in between. Logical flow optimization bears similarities with database query optimization but the problem is actually more complex because (i) query optimizers cannot consider the dependency constraints between tasks that appear in data flows, (ii) the tasks in a flow execution graph do not necessarily belong to a specific set of operators with clear semantics, and (iii) the optimization objective is not limited to performance. Overall, data flow optimization can be inspired by query optimization techniques that perform structure reformations, such as reordering and introducing new tasks in an existing flow, but it cannot fully rely on them. For example, in [9] ad-hoc query optimization methodologies are employed for optimizing the flow execution plan by reordering and introducing filtering tasks. Other logical flow optimization proposals consider swapping re-orderable flow activities, merging tasks, splitting, and so on, in order to generate new flow execution plan alternatives for ETL flows [14]. Additional narrower proposals include task consolidation for reducing the overall execution time [16, 5].

At the physical level, a wide range of implementation aspects need to be specified so that the flow can be executed. The most significant of them include the choice of the exact implementation alternative for each task, the selection of the execution engine to run tasks, scheduling details, the manner in which data is transmitted between tasks, decisions as to whether the tasks are executed in a pipelined or step-wise fashion, and so on. For all those aspects, several techniques have been proposed, which assume that the flow has been already optimized at the logical level. For example, [17] proposes resource allocation algorithms and heuristic techniques taking into account constraints, such as cost optimization, user-specified deadline and workflow partitioning according to assigned deadlines, while a set of optimization algorithms for scheduling flows based on deadline and time constraints is analyzed in [2].

So far, the existing flow optimization methodologies tend to focus either on the physical or the logical level, without

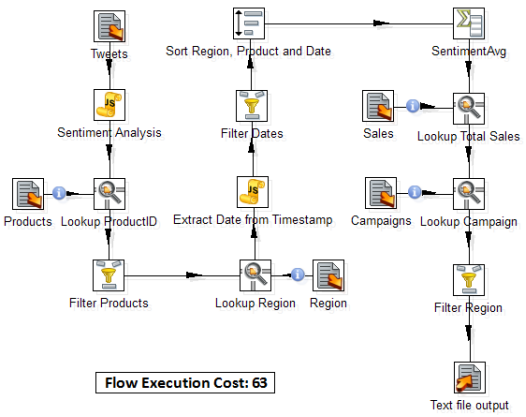

Figure 1: A real-world analytic data flow.

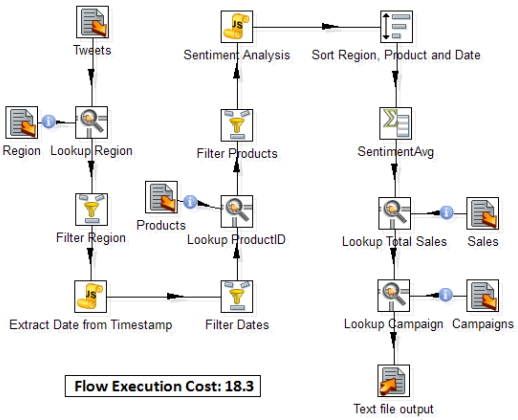

Figure 2: The optimized version of the data flow.

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	173	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

Table 1: The cost and selectivities values.

providing a holistic optimization proposal for both levels. In this paper, we argue that the existing techniques regarding logical flow optimization, although they are interesting, they are inadequate. An optimal optimization solution referring at the physical level, which is based on a suboptimal logical plan, yields an overall suboptimal execution. We restrict ourselves to one of the simplest cases and we show that even for that case, the state-of-the-art can be significantly advanced. More specifically, we focus on *single-input single-output (SISO)* data flows, for which only the sequence of tasks needs to be specified so that all dependency constraints are respected and the single optimization criterion is the minimization of the sum of task execution times. We provide a real use case that demonstrates the need for more advanced optimization (Sec. 2), and we discuss the gap of existing optimization proposals and our proposal for near-optimal flow execution plans (Sec. 3).

2. MOTIVATIONAL EXAMPLE

A real data flow that processes free-form text data from Twitter commenting on products in order to compose a dynamic report that associates sales with marketing campaigns is shown in Figure 1. The flow is implemented with the help of the Pentaho Data Integration tool (`http://kettle.pentaho.com`). As shown in the figure, this data flow consists of 13 tasks (or nodes or activities) with a single streaming source that outputs tweets on products and a sink task where the resulting report is stored. During processing, it accesses four static sources (databases) through lookup operations. The remainder 7 tasks perform various operations, such as computing a single sentiment value for the products that are mentioned in the tweet (*Sentiment Analysis*), filtering according to several criteria including product type, and date, transformation of the timestamp text data to date and ad-hoc aggregation (*SentimentAvg* that averages sentiment values over each region, product, and date). At the final stage, the user has the option to narrow down the report in order to focus on a specific region. Table 1 shows the selectivity and cost values computed for a specific dataset of 1M records. 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 4 filtering tasks, while the rest do not modify the number of records (note that in general, selectivities may be higher than 1). For this flow, there are 38% precedence constraints (PCs, not presented in detail due to space constraints), where a fully constrained flow with n tasks and 100% PCs has $\frac{n(n-1)}{2}$ constraints and no equivalent ordering alternatives.

We run an exhaustive algorithm and we find the optimal flow for those statistics and precedence constraints, as shown in Figure 2. We also apply the best-performing approximate heuristic to date, which is proposed in [14]. Both exhaustive and approximate solutions are discussed in the next section. As we can see, the exhaustive optimization moves filtering according to 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 total execution times of the initial, optimal and heuristic (i.e., approximately optimized) flow designs are 63, 18.3 and 36.5 seconds, respectively when run on a Intel Core i5 machine. This is a representative example of a 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 2. 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.

3. OPTIMIZATION SOLUTIONS

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 [3]. Here, we discuss the inherently non-scalable exhaustive solutions, the state-of-the-art heuristics and our proposal for flow optimization which improves on the latter heuristics.

3.1 Accurate (exhaustive) algorithms

Backtracking algorithm: The *Backtracking* algorithm, as presented in [8], finds all the possible execution plans generated after reordering the tasks of a given data flow preserving the precedence constraints (PCs). The algorithm enumerates all the valid sub-flow plans after applying a set of recursive calls on these sub-flows and runs in $O(n!)$ time.

Dynamic Programming (DP): The rationale of the *DP* algorithm extends its query optimization counterpart to calculate 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 PCs. The time complexity of *DP* is $O(n^2 2^n)$.

Topological Sorting (TS): Contrary to query optimization, we have found that enumerating all orderings that meet the dependency constraints is a viable option for flows with numerous PCs, although in the worst case the *TS* algorithm runs in $O(n!)$. Due to space limitations, we do not give full details, but as shown in the experiments below, a variant based on [15] can significantly outperform the two other exhaustive approaches mentioned above in terms of optimization time (overhead).

3.2 Approximate (heuristic) algorithms

The state-of-the-art heuristics for flow task ordering is summarized below. Preliminary results regarding their efficiency in optimizing flows is provided in [10].

Swap: The *Swap* algorithm, proposed in [14], compares the cost of the existing execution plan against the cost of the transformed plan, if we swap two adjacent tasks provided that the PCs are always satisfied. This check is performed for every pair of adjacent tasks. The complexity is $O(n^2)$. **GreedyI:** The *GreedyI* algorithm is based on a typical greedy approach, which builds the flow execution plan incrementally. In each step, it adds at the end of the partial plan the activity with the maximum rank $\frac{1-selectivity}{cost}$ among those for which all the prerequisite tasks have been already added. The time complexity is $O(n^2)$. **GreedyII:** The rationale of the *GreedyII* algorithm 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). The algorithm is presented in [12]. **Partition:** The *Partition* algorithm, in each step, detects the set of tasks that can be added based on the PCs. For that set, it exhaustively finds the optimal sub-solution, and then proceeds to the next set until all activities have been added. It runs in $O(n!)$ time in the worst case.

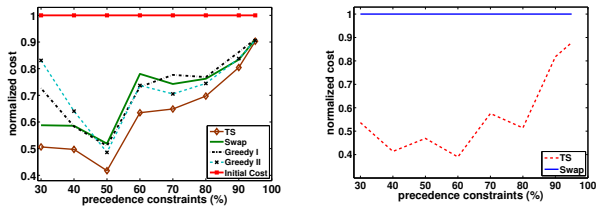

Figure 3: Average (left) and maximum (right) improvements of exhaustive solutions

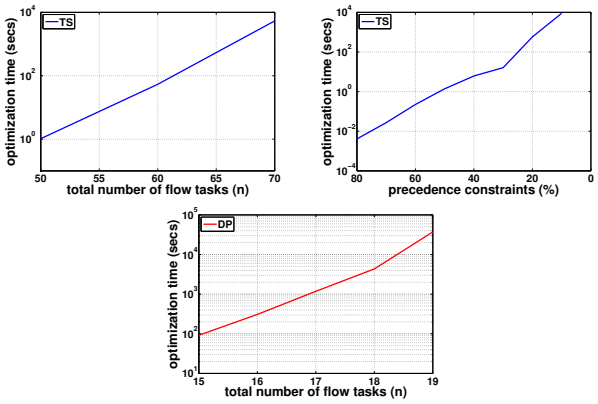

Figure 4: Optimization overhead for *TS* and *DP*

3.3 Pros and cons of exhaustive solutions

We conducted a set of experiments to further support the observation in Section 2 that the existing heuristics fail to approximate the optimal solution. We examined randomly generated data flows consisting of $n = 10, 15, \ldots, 25$ tasks, selectivity values $sel \in (0, 2]$ where selectivities higher than 1 denote increase in the task output, $cost \in [1, 10]$ and 30%-95% PCs. The flows were executed on an Intel Core i5 machine and all experiments were repeated 20 times. The results show that the performance improvement derived by the application of exhaustive algorithms is significantly high for small flows, as they provide the optimal solution. For example, Figure 3(left) presents the results for flows with 15 tasks; the accurate algorithms, such as *TS*, can have up to 59% better performance improvement compared to a random initial flow that just respects the PCs. On average, the best performing heuristic is *Swap*. Figure 3(right) shows the maximum normalized difference between *Swap* and *TS*, which can reach up to 61% in favor of the latter.

However, exhaustive solutions are inapplicable for medium and large flows, and/or few PCs. As shown in Figure 4(top), *TS* cannot scale in either the number of the flow tasks or the PCs. *DP*'s overhead does not depend on the PCs and, as shown in Figure 4(bottom), this algorithm is not practical for flows with more than 18 tasks. *Backtracking* is less scalable than *TS* as well, with higher overhead by a factor between 46 and 62 on average.

3.4 Our optimization proposal

In the quest of finding an efficient optimization solution for our problem, as our starting point, we chose the well-known KBZ query optimization algorithm for join ordering in [11]. That algorithm is of low complexity $O(n^2)$ and can consider PCs that can be represented as a rooted tree. The rationale is to consider the rank of each task and order tasks by their

rank value; if this is not possible due to PCs, then tasks are merged and the rank values are updated accordingly. The evaluation results (not presented here in detail) show that, when applicable, this approach can be dozens of times less expensive than *Swap* and the other heuristics. However, allowing only tree-shaped PC graphs implies that there is no task with more than one independent prerequisite activity and the percentage of PCs is very low and decreases with the number of tasks (e.g., less than 10% for a 100-node flow); both cases do not occur frequently in practice.

So, to combine KBZ's efficiency and support for generic flows, we reduce our problem to that of transforming the DAG that typically represents PCs (an edge from one task to another denotes that the former must precede) into an acyclic one after removing edge directions. We initially propose a simple heuristic algorithm *RO-I* (for rank ordering) that, omitting implementation details, transforms the graph of PCs into an acyclic one by removing incoming edges with no maximum rank, if a task has more than one incoming edge. Then it applies the KBZ algorithm and is followed by a post-processing phase, where any resulting PC violations are resolved by moving tasks upstream if needed as prerequisites for other tasks placed earlier. That heuristic is simple but does not always behave well. We have investigated another approach, termed as *RO-II*, which detects paths in the PC graph that share an intermediate source and sink and merges them to a single path based on their rank values. In that way, both all PCs are preserved and the rationale of rank ordering is kept at the expense of implicitly examining fewer re-orderings. As such, these local optimizations do not guarantee a globally optimal solution.

Preliminary evaluation results are shown in Figures 5 and 6. We can see that the average improvements of *RO-II* over *Swap* can be significant, whereas *RO-I* in some cases outperforms *RO-II* but in others is significantly worse. For isolated runs, we have observed that *Swap* can be up to 7 times more expensive.

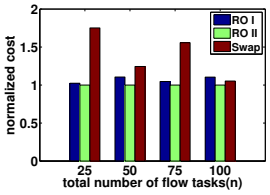

Figure 5: 20% PCs.

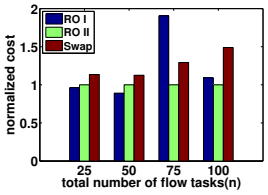

Figure 6: 50% PCs.

4. CONCLUDING REMARKS

The fact that existing logical optimization techniques are inadequate to provide a (near) optimal solution even for small flows implies that, even after applying the most advanced physical optimization techniques, the execution performance is suboptimal since the latter techniques depend on the former. More research is needed (i) for deciding the sequence of the flow tasks and (ii) for building more holistic approaches that consider additional factors, such as merging and splitting tasks and physical implementation details. For item (i), our proposal is promising and the results provided here provide strong insights in its ability to fill the gap between exhaustive non-scalable solutions and existing heuristics; however more research and thorough analysis and evaluation is needed for rank-ordering-based solutions.

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.

5. REFERENCES

[1] D. Abadi *et al.* The beckman database research self-assessment meeting. Technical report, 2013.

[2] S. Abrishami, M. Naghibzadeh, and D. H. Epema. Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds. *Future Generation Computer Systems*, 29(1):158 – 169, 2013.

[3] J. Burge, K. Munagala, and U. Srivastava. Ordering pipelined query operators with precedence constraints. Technical Report 2005-40, Stanford InfoLab, 2005.

[4] S. Chaudhuri, U. Dayal, and V. Narasayya. An overview of business intelligence technology. *Commun. ACM*, 54:88–98, 2011.

[5] R. Dewan, A. Seidmann, and Z. Walter. Workflow optimization through task redesign in business information processes. In *HICSS*, pages 240–252. IEEE Computer Society, 1998.

[6] R. Halasipuram, P. M. Deshpande, and S. Padmanabhan. Determining essential statistics for cost based optimization of an etl workflow. In *EDBT*, pages 307–318, 2014.

[7] S. Holl, O. Zimmermann, M. Palmblad, Y. Mohammed, and M. Hofmann-Apitius. A new optimization phase for scientific workflow management systems. *Future Generation Comp. Syst.*, 36:352–362, 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] G. Kougka and A. Gounaris. On optimizing work ows using query processing techniques. In *SSDBM*, pages 601–606, 2012.

[10] G. Kougka and A. Gounaris. Declarative expression and optimization of data-intensive flows. In *DaWaK*, pages 13–25, 2013.

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

[12] N. Kumar and P. S. Kumar. An efficient heuristic for logical optimization of etl workflows. In *BIRTE*, volume 84 of *Lecture Notes in Business Information Processing*, pages 68–83. Springer, 2010.

[13] E. S. Ogasawara, D. de Oliveira, P. Valduriez, J. Dias, F. Porto, and M. Mattoso. An algebraic approach for data-centric scientific workflows. *PVLDB*, 4:1328–1339, 2011.

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

[15] Y. L. Varol and D. Rotem. An algorithm to generate all topological sorting arrangements. *The Computer Journal*, 24(1):83–84, 1981.

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

[17] Z. Xiao, H. Chang, and Y. Yi. Optimization of workflow resources allocation with cost constraint. In *Proc. of the 10th Int. Conf. on Computer supported cooperative work in design*, pages 647–656, 2007.