Pasta: A Cost-Based Optimizer for Generating Pipelining Schedules for Dataflow DAGs

XIAOZHEN LIU, University of California, Irvine, USA
YICONG HUANG, University of California, Irvine, USA
XINYUAN LIN, University of California, Irvine, USA
AVINASH KUMAR, University of California, Irvine, USA
SADEEM ALSUDAIS*, King Saud University, Saudi Arabia
CHEN LI, University of California, Irvine, USA

Data analytics tasks are often formulated as data workflows represented as directed acyclic graphs (DAGs) of operators. The recent trend of adopting machine learning (ML) techniques in workflows results in increasingly complicated DAGs with many operators and edges. Compared to the operator-at-a-time execution paradigm, pipelined execution has benefits of reducing the materialization cost of intermediate results and allowing operators to produce results early, which are critical in iterative analysis on large data volumes. Correctly scheduling a workflow DAG for pipelined execution is non-trivial due to the richer semantics of operators and the increasing complexity of DAGs. Several existing data systems adopt simple heuristics to solve the problem without considering costs such as materialization sizes. In this paper, we systematically study the problem of scheduling a workflow DAG for pipelined execution, and develop a novel cost-based optimizer called Pasta for generating a high-quality schedule. The Pasta optimizer is not only general and applicable to a wide variety of cost functions, but also capable of utilizing properties inherent in a broad class of cost functions to improve its performance significantly. We conducted a thorough evaluation of developed techniques on real-world workflows and show the efficiency and efficacy of these solutions.

CCS Concepts: • **Information systems** → **Query optimization**; *Data analytics*; *Computing platforms*; • **Software and its engineering** → *Data flow architectures.*

Additional Key Words and Phrases: scheduler, workflow, pipelined execution, data engine, physical plan

ACM Reference Format:
Xiaozhen Liu, Yicong Huang, Xinyuan Lin, Avinash Kumar, Sadeem Alsudais, and Chen Li. 2024. Pasta: A Cost-Based Optimizer for Generating Pipelining Schedules for Dataflow DAGs. *Proc. ACM Manag. Data* 2, 6 (SIGMOD), Article 248 (December 2024), 26 pages. https://doi.org/10.1145/3698832

1 Introduction

Many data analytics tasks are conducted as workflows [1, 22, 27] represented as directed-acyclic graphs (DAGs) of operators. With the recent advances in machine learning techniques, often incorporated as user-defined functions [8, 29], and the growing popularity of data science in many disciplines, there is an increasing complexity of dataflow DAGs. This increase is reflected in the

*The author did most of her work of the paper while being a PhD candidate at UC Irvine.

Authors' Contact Information: Xiaozhen Liu, xiaozl3@ics.uci.edu, University of California, Irvine, Irvine, CA, USA; Yicong Huang, yicongh1@ics.uci.edu, University of California, Irvine, Irvine, CA, USA; Xinyuan Lin, xinyual3@ics.uci.edu, University of California, Irvine, Irvine, CA, USA; Avinash Kumar, avinask1@uci.edu, University of California, Irvine, Irvine, CA, USA; Sadeem Alsudais, salsudais@ksu.edu.sa, King Saud University, Riyadh, Saudi Arabia; Chen Li, chenli@ics.uci.edu, University of California, Irvine, Irvine, CA, USA.

ACM 2836-6573/2024/12-ART248
https://doi.org/10.1145/3698832

number of operators and edges, the variety of operators, and the structural intricacy of the DAGs. This trend, combined with larger data volumes, poses great challenges to dataflow systems in terms of both efficiency and scheduling.

As an example, consider an image-analysis workflow shown in Figure 1. It reads 25,000 images from a list of files, uses 30% of them (done by the Split operator) to train a machine learning (ML) model using operator v_6, and utilizes the model to classify the remaining 70% of the images[1]. The inferred results are aggregated and then visualized using operator v_9. Several operators generate gigabytes of data due to the large number and size of images (4MB each). Such a workflow arises often in a data science project, which focuses on model tuning and development, as opposed to a production setting. For instance, in our analysis of 6,000 workflows (Section 6), 48% of ML-related workflows include training and inference operators in the same DAG.

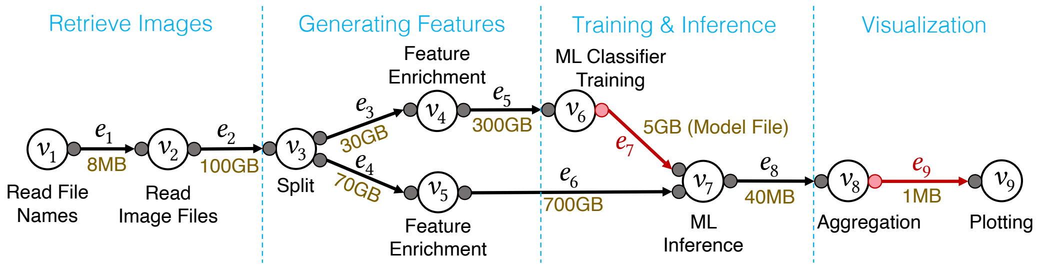


Figure 1. A dataflow DAG for ML-based image analysis, which does a 30/70 split of images for training and inference. The number on each edge is the size of its intermediate results. A red port/edge means a *blocking* port/edge, which will be explained in Section 2.

Two execution strategies. One way to execute this workflow is *operator-at-a-time*, i.e., we follow a topological order of the operators and run them one by one. After running an operator, we materialize its output data, either in memory or on disk. For each operator, all of its input data should be available before it starts its computation. The main drawback of this approach is the large sizes of intermediate results and the corresponding materialization overhead. We could solve the problem by doing *pipelining*. For example, while the Split operator reads the images on the fly, it immediately sends its produced results to the downstream operators v_4 and v_5 without materializing these images. This pipelined execution can not only reduce the materialization cost but also allow operators to produce results early. In particular, once the ML model is trained in operator v_6, the user can see partial results of the ML Inference operator v_7 from the classified images, without waiting for all the images to be processed. Producing results early by operators is especially beneficial in data science, which is known to be iterative and requires users to constantly modify a workflow based on initial results from operators.

Scheduling problems in pipelined execution. A main issue in scheduling this workflow to do a pipelined execution is the ML training operator v_6. In particular, this operator needs to receive all its training instances before producing a model. (We represent this behavior of the operator by marking its output port and the corresponding output edge as red.) If we let all the operators start their execution at the same time to do pipelining, the ML inference operator v_7 has to wait for the operator v_6 to complete its training, which may take minutes or even hours. During this period, v_7 has to buffer a large number of images, which again introduces a high materialization cost.

One may wonder whether we can avoid this scheduling problem by dividing the workflow into two. The first workflow uses v_1, v_2, and v_3 from Figure 1 to produce two sets of images. The

[1]This example is based on a real workflow in the domain of neuroscience with simplification for presentation purposes.

second one reads the two image sets and uses the remaining operators to continue the analysis. (Interestingly, the aforementioned scheduling problem disappears in the second workflow as its structure is a simple tree rather than a non-tree DAG.) This approach is not ideal in many applications where the users, especially domain experts, prefer to have a single workflow to conduct the entire pipeline for the benefit of easy understanding and efficient management of self-contained workflows. Our analysis of real-world workflows shows that 52% of them are non-tree DAGs (Section 6).

Existing scheduling solutions and limitations. This scheduling problem was already observed in push-based data-processing systems that support pipelined execution (e.g., Hyracks [3]), which often adopt heuristic-based solutions. For the running example, Flink [2] (in its batch mode) and Hyracks add a materialization step on the edge e_6 immediately before operator v_7. While this heuristic does solve the aforementioned problem, it needs to materialize 700GB of data, which is much worse than another plan that materializes edge e_4 with a much lower cost of 70GB. This example shows that a good materialization choice should be *cost-based.*

Challenges. When developing a general cost-based solution for this problem, a main challenge is the complexity of workflows. For instance, Figure 2 shows a real dataflow from Alteryx [1] with more operators and edges, including blocking edges marked in red. Many real-world workflows are even more complicated, easily with hundreds of operators and edges. Generating an optimal execution order (i.e., "schedule" in our context) efficiently on such complex workflows can be computationally expensive. In addition, when formulating an optimization problem, we notice that the literature lacks a clear description of the relationships between common concepts such as *blocking*, *materialization*, and *pipelining* and modules of a workflow such as operators, ports, and edges, as well as their execution and scheduling. We need to develop a formal framework to present these relationships and clearly define a scheduling-based optimization problem.

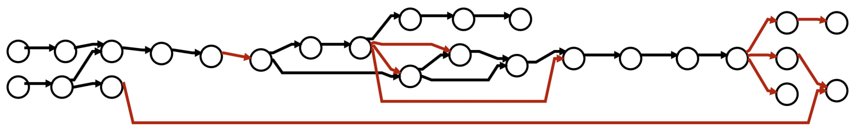


Figure 2. A more complicated dataflow. Red edges are blocking, and operator details are omitted for brevity.

In this paper, we systematically study the problem of generating an efficient execution order for a dataflow DAG in a pipelining setting. We develop a novel optimizer for data-processing systems called "Pasta," as shown in Figure 3. The optimizer is not only general and applicable to a wide variety of cost functions, but also capable of utilizing common conditions satisfied by many cost functions to improve its performance significantly. We make the following contributions. In Section 2, we give an overview of the optimizer, and formally define the optimization problem. In Section 3 we identify several interesting properties of an optimal execution order. In Section 4, we develop a top-down search framework to find an execution order, and show how to utilize these properties to improve the search performance. In Section 5 we show a bottom-up framework to do a search in the opposite direction, discuss how to obtain costs for the Pasta optimizer, and extend the results to other cost functions. In Section 6, we report experimental results of a thorough evaluation of the techniques using real-world dataflows to show their efficiency and efficacy.

1.1 Related Work

Dataflow systems. The studied problem arises in the context of a pipelined-execution model, which is adopted in systems such as Flink [2], Hyracks [3], and Amber [16]. It also assumes a push-based engine that executes a physical plan as a DAG as opposed to a tree [1–3, 13, 22, 26].

In pull-based engines such as Apache Spark [31], a physical plan typically is a tree of operators, where the studied problem becomes less challenging [17].

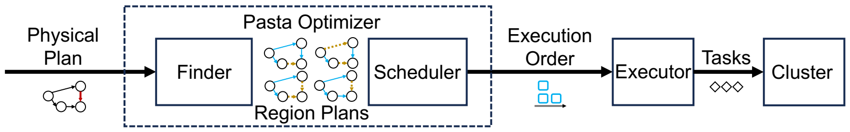


Figure 3. Overview of our proposed Pasta optimizer in a data-processing system.

Dataflow schedulers. Improving the scheduling of dataflow DAGs has been studied extensively in the literature. For example, [10, 28] studied optimizations using DAG-aware task scheduling. Existing studies mainly focused on using a single region plan with a fixed choice of pipelining or materialization for an edge. Pasta is more powerful as it considers multiple region plans in scheduling (using the Finder module in Figure 3), and treats the process of generating execution orders for a region plan as a black box. As a result, existing scheduling optimizations can be adopted as the Scheduler in Pasta.
Cluster resource managers. These managers, such as Apache Mesos [12], Apache Hadoop YARN [25], Swarm [7], and Kubernetes [15], correspond to the Executor module in Figure 3. They focus on how to allocate cluster resources. In the context of dataflow systems, these managers are used primarily for orchestrating tasks submitted by a dataflow system, and the scheduling usually takes sets of stages [3, 31] or pipelined regions [20] of operators as the input, which correspond to the output of Pasta.
Optimizations on pipelining. Pipelining techniques in database systems support inter-operator parallelism. Early works focused on optimizing pipelining for query-plan trees and for specific operators [4, 11, 19]. The studies in [5, 23] considered pipelining in the context of multi-query optimization in DBMS, where DAGs are common. The work [5] did not consider operators with more than one output port, and did not distinguish between blocking/non-blocking ports in their problem formulation. For example, consider a workflow DAG with the same structure as in Figure 1 but without any blocking port. An optimal execution order generated in [5] still materializes at least two edges, while Pasta allows an execution order that pipelines all the edges. [23] studied a deadlock problem when pipelining is used in multi-query optimization. Another related work is operator fusion [24], which combines multiple operators in a physical plan. This technique is orthogonal to our problem, as Pasta can treat a chain of fused operators as a single operator in a physical plan.

2 Problem Formulation in Pasta

Figure 3 shows a data-processing system that uses Pasta, which takes a physical plan as input, considers region plans, and generates an execution order to be executed by an Executor on a compute cluster. A system that intends to use Pasta typically (1) processes bounded input data (as opposed to stream processing); (2) uses a push-based execution model in the Executor; and (3) supports execution of DAG-based physical plans with data-processing operators and without control operators [9] such as if-else switches and for-loops. In this section, we formulate an optimization problem in Pasta to schedule a physical plan.

2.1 Physical Plans and Blocking Ports

A *physical plan* is a directed acyclic graph (DAG) denoted as $P = (V, E)$, where each vertex in V is an *operator* that represents a data-processing unit. Each edge $e = (v_x, v_y) \in E$ is a *physical edge*, which is a directed connection from an output port of an operator v_x to an input port of an operator v_y. A physical plan can be created by an analyst using a GUI interface [18], or generated from a logical plan by a compiler [21].

Definition 2.1 (Blocking port). An output port of an operator is *blocking* if it produces output tuples only after all of this operator's input ports have received their tuples. Otherwise, the output port is called *non-blocking*.

It is often easier to refer to an edge than a port. For easy presentation, we call an edge *blocking* or *non-blocking* if its sending port is *blocking* or *non-blocking*, respectively. Correspondingly, we mark a blocking edge red to make this property more visible. Figure 4 shows an example physical plan that tests the performance of an ML model. Operator v_5 requires receiving all its input tuples to train a model, thus its output port is blocking and e_6 is a blocking edge. Operator v_6's output port that produces evaluation metrics is also blocking since the operator needs both of its input ports to fully receive their data before producing the metrics. Another output port of v_6 produces the prediction results for each input tuple of the testing set. Since v_6 produces predictions tuple-by-tuple without waiting to receive all the tuples from e_5, this port is non-blocking. Finally, the prediction results are connected to v_8, which selects the wrong predictions for further analysis. Notice that edges e_2 and e_3 are connected to the same output port of operator v_2, which is non-blocking. As a result, these edges must have the same non-blocking property.

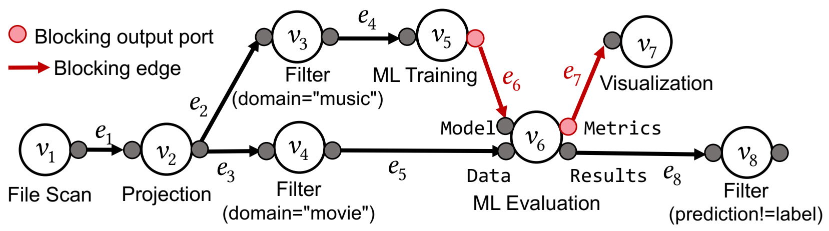


Figure 4. A physical plan P_1 with blocking ports and their edges marked in red.

2.2 Region Plans with Pipelined/Materialized Edges

Definition 2.2 (Region plan). A *region plan* [2] of a physical plan $P = (V, E)$, denoted as $U = (P, \phi)$, specifies a mapping ϕ from the set of edges E to a set of two labels {Pipelined, Materialized}. The two labels specify the method of data transfer on an edge:

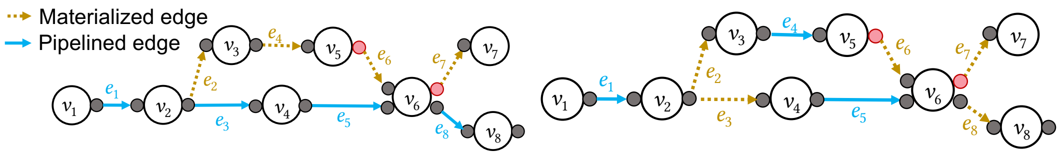
- *Pipelined Edge*: An edge $e_i = (v_x, v_y)$ is called *pipelined* if v_x passes a tuple to v_y as soon as the output port of v_x connected to e_i produces a tuple.
- *Materialized Edge*: An edge $e_i = (v_x, v_y)$ is called *materialized* if v_x saves all its output tuples (e.g., in memory or on disk), which will later be consumed by v_y.

For simplicity, in the rest of the paper, we use $U = (V, E, \alpha_U, \beta_U)$, or simply (U, α_U, β_U) to denote a region plan for a physical plan $P = (V, E)$, where α_U is the set of pipelined edges and β_U is the set of materialized edges.

Figure 5a shows a region plan U_1 for the physical plan in Figure 4. Edges e_6 and e_7 need to be materialized because their connected output ports in U_1 are blocking. Edge e_4 is non-blocking in U_1 and is materialized in U_1. Interestingly, edge e_2 is materialized and e_3 is pipelined, even though

[2]The name "region" will be defined formally in Section 2.3.

they are connected to the same non-blocking output port of v_2. Figure 5b shows another region plan U_2 for the same physical plan.



(a) A region plan U_1. (b) Another region plan U_2.

Figure 5. Two region plans for the physical plan P_1.

In general, consider a blocking edge $e_i = (v_x, v_y)$ in a physical plan. If e_i is pipelined in a region plan, then whenever v_x produces a tuple, v_y should be ready to consume it. In terms of scheduling, this means that they need to run in parallel. However, as e_i is blocking, v_x does not produce anything until it has processed all its input data. During this period, v_y is idle, which could waste a significant amount of system resources. To avoid this problem, Pasta requires a blocking edge to always be materialized. Table 1 shows the compatibility of physical-plan edges and region-plan edges. Table 2 summarizes the concepts defined so far.

Table 1. Compatibility of physical-plan edges and region-plan edges.

Physical plan / Region plan	Blocking edge	Non-blocking edge
Materialized Edge	Compatible	Compatible
Pipelined Edge	**Incompatible**	Compatible

Table 2. Concepts related to pipelined execution.

Level	**Concept**	**Description**
Physical plans	Operator	Basic computing unit in a workflow.
	Blocking port	An output port of an operator that does not produce anything until the operator has received all its input data.
	Non-blocking port	An output port that is not blocking.
	Blocking/non-blocking edge	Derived from the blocking property of the sending port of an edge.
Region plans	Pipelined edge	Transferring data between two operators by passing the tuple as soon as it is produced.
	Materialized edge	Saving all intermediate tuples before transferring them to the downstream operator.

2.3 Regions and Region Graphs

Given a region plan, Pasta decides an order of executing the operators based on the notion of *regions*.

Definition 2.3 (Region). Given a region plan $U = (V, E, \alpha_U, \beta_U)$, a *region* R is a weakly-connected sub-DAG of U such that all the edges in R are pipelined.

Each pipelined edge $e = (v_x, v_y)$ in a region plan requires operators v_x and v_y to start together. Pasta ensures this requirement is satisfied for all pipelined edges. A region includes a set of all the operators that must start processing together. Given a region plan $U = (V, E, \alpha_U, \beta_U)$, for an operator $v_i \in V$, we denote the region that v_i belongs to as $R_U(v_i)$.

Figure 6a shows the regions of U_1 in Figure 5a. Each of the regions R_2, R_3, and R_4 contains one operator, which is not connected to any other operator by a pipelined edge. The region of v_4 is R_1, i.e., $R_{U_1}(v_4) = R_1$.

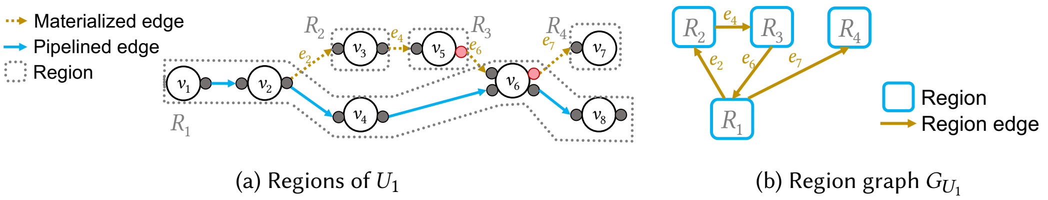


Figure 6. The regions and region graph of U_1 in Figure 5a.

Figure 7a shows the regions of the region plan U_2 in Figure 5b.

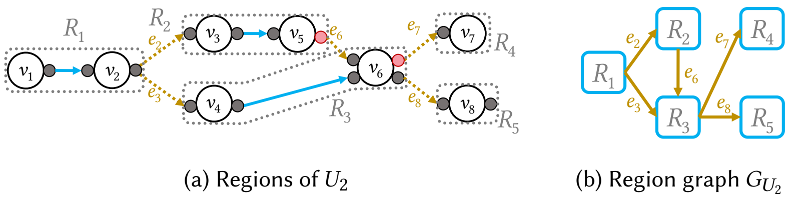


Figure 7. The regions and region graph of U_2 in Figure 5b.

Region-based execution model. Pasta runs a region plan region by region. In particular, the *start* of a region requires all its operators to start processing, allowing pipelined execution among the operators in the same region. The *completion* of a region means that all its operators have finished processing and produced their results. For each materialized edge $e = (v_x, v_y)$ in the region plan, the region $R(v_y)$ can only start execution after $R(v_x)$ has completed execution. In this region-based execution model, each materialized edge $e = (v_x, v_y)$ in a region plan U derives a time dependency between the regions $R_U(v_x)$ and $R_U(v_y)$. We denote this dependency as a *region edge*. We then have a graph of dependencies between regions.

Definition 2.4 (Region graph). The *region graph* G_U for a region plan U is a directed graph $G_U = (V_{G_U}, E_{G_U})$, where V_{G_U} is the set of regions of U, and E_{G_U} is the set of region edges derived from the materialized edges of U, i.e., each materialized edge $e_i = (v_x, v_y) \in \beta_U$ corresponds to a region edge $e_i = (R_U(v_x), R_U(v_y)) \in E_{G_U}$. Each region edge $e_i = (R_x, R_y)$ means R_x should finish before R_y starts.

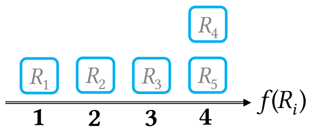
For example, Figure 6b and Figure 7b show the region graphs of U_1 and U_2, respectively. A region edge such as e_3 in G_{U_2} (Figure 7b) means R_1 must finish before R_3 can start, i.e., all operators in R_1 (v_1 and v_2) must finish processing before any operator in R_3 (v_4 and v_6) can start processing. The region graphs G_{U_1} and G_{U_2} capture all such temporal constraints between the regions of U_1 and U_2, respectively. Note there is a cycle in G_{U_1}, and we will explain its meaning next.

2.4 Schedulability of Region Plans

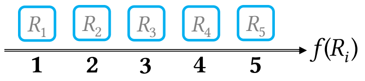
Definition 2.5 (Execution order). For a region plan U, an *execution order* S of U is a *ranking* of the regions V_{G_U}, i.e., a many-to-one mapping from the regions of U to a set of rank numbers (a.k.a. ranks) $f : V_{G_U} \rightarrow \mathbb{N}^+$, such that for any region edge $e_i = (R_x, R_y) \in E_{G_U}$, $f(R_x) < f(R_y)$.

We use rankings instead of total orders to define execution orders because Pasta allows two regions that are not reachable from each other via region edges to start in parallel. For example, in Figure 7b, R_4 and R_5 can have the same rank.

We call a region plan with an acyclic region graph *schedulable.* Otherwise, when the region plan has a cyclic region graph, it is impossible to generate an execution order for this region plan, thus it is *unschedulable.* Note schedulability is a property of the region plan instead of the physical plan. The region graph G_{U_1} in Figure 6b is cyclic, thus U_1 is unschedulable. The region graph G_{U_2} in Figure 7b is acyclic, thus U_2 is schedulable. Figure 8 shows two execution orders for U_2, namely S_1 and S_2. In execution order S_1, regions R_4 and R_5 are started together, while in execution order S_2, region R_5 can only start after R_4 has finished.



(a) Execution order S_1.



(b) Execution order S_2.

Figure 8. Two execution orders for the region plan U_2.

Goal of Pasta: Given a physical plan as a DAG, generate an optimal execution order based on a cost function of execution orders.

We can show that two region plans do not share execution orders. Therefore, to enumerate execution orders for a physical plan, we can first enumerate all its region plans (performed by the Finder module), then generate an execution order for each region plan (performed by the Scheduler module). Figure 9 shows this enumeration process using a *region-plan plane* and an *execution-order plane.* To find a good execution order, the Scheduler needs to choose one from the execution orders of a region plan. There can be an exponential number of execution orders for a region plan. Many systems [2, 31] only consider a limited set of execution orders, e.g., a total order of the regions. There are studies [10, 28] on DAG-based scheduling that can be utilized by the Scheduler module in Pasta to optimize the process of generating an execution order given a region plan. We treat this process as a black box in the rest of the discussion.

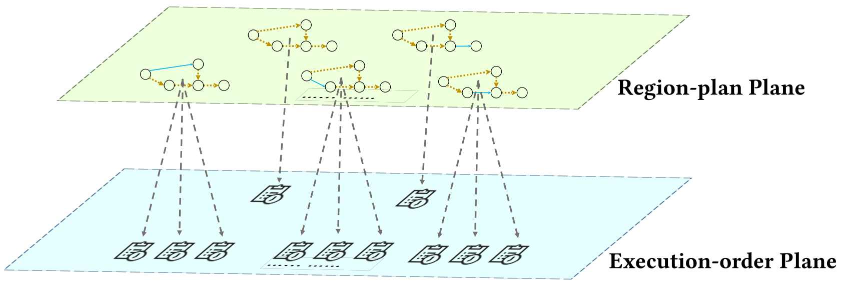


Figure 9. The region-plan plane for a physical plan and the corresponding execution-order plane.

2.5 Costs of Execution Orders and Region Plans

There can be diverse optimization goals, e.g., minimizing materialization sizes, minimizing CPU costs, reducing the wall-clock time of a workflow, etc. To maintain the generality of Pasta, we assume a *cost function* that computes the cost of running an execution order. The objective and quality of the given cost function are not the main focus of Pasta. Given a cost function, the cost of a region plan is based on the costs of its execution orders.

Definition 2.6 (Cost of a region plan). The cost of a region plan U is $Cost(U) = \min_{S_i \in \mathcal{S}(U)} Cost(S_i)$, where $\mathcal{S}(U)$ is the set of execution orders for U considered by the Scheduler. If U is unschedulable, its cost is infinite.

For easy presentation, we first study the problem by considering a simple and commonly used cost function.

Cost Function A (total materialization size). The cost of an execution order is the sum of the sizes of its materialized edges.

Using this cost function, all the execution orders of a region plan have the same cost. Thus the cost of a region plan is also the sum of the materialization sizes of its materialized edges. We will extend the results to other cost functions in Section 5.

In the rest of the paper, we denote an optimal schedulable region plan as "OSRP."

3 Properties of Optimal Schedulable Region Plans

In this section, we present several interesting properties of OSRP's, which will be used to improve the performance of a search method to generate an optimal execution order.

3.1 Properties of Chains

Definition 3.1 (Chain). A *chain* in a physical plan DAG is a path such that each of its operators (except the first and the last) is connected only to operators on the path.

A chain that is not a proper sub-path of any other chain is called a *maximal chain.* Figure 10 shows example chains in a physical plan. H_4 is not maximal as it is a proper sub-path of another chain H_2, which is maximal. Paths H_1 and H_3 are two other maximal chains.

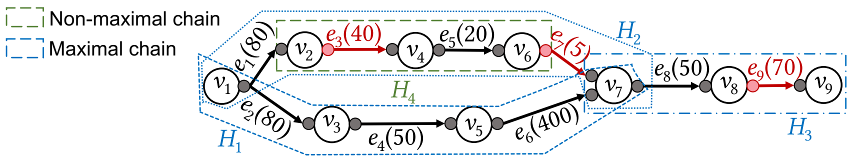


Figure 10. A physical plan P_3 with three maximal chains. Edge costs are shown in the parentheses.

The following results show that we only need some of the non-blocking edges on a chain to be materialized to maintain the schedulability of a region plan.

LEMMA 3.2. *For a chain H in a physical plan P, if a schedulable region plan U has more than one materialized edge on H, then changing each materialized edge on H to pipelined produces another schedulable region plan.*

LEMMA 3.3. *If a chain H in a physical plan P contains a blocking edge, each optimal schedulable region plan for P must pipeline all the non-blocking edges on H.*

For example, for the physical plan P_3 in Figure 10, the chain H_2 has two blocking edges e_3 and e_7, which must be materialized in each region plan. Thus an OSRP for this physical plan must pipeline the other edges on this chain, namely e_1 and e_5. Similarly, on the chain H_3, an OSRP must pipeline e_8.

Lemma 3.4. *If a chain H in a physical plan P does not contain a blocking edge, each optimal schedulable region plan for P has at most one non-blocking materialized edge on H.*

For example, the chain H_1 in the physical plan P_3 is a chain without a blocking edge. Then in each OSRP, there cannot be more than one materialized edge on this chain. As a consequence, there is no need to consider region plans with two or more materialized edges on H_1.

Corollary 3.5. *Given a physical plan P, for a maximal chain H that does not contain a blocking edge, if an optimal schedulable region plan includes a materialized edge e in H, then e has the minimal cost among all the edges of H.*

For instance, for the physical plan P_3, if an OSRP has an edge materialized on H_1, then this edge must be the one with the lowest cost, i.e., e_4.

3.2 Properties of Clean Edges

Next, we identify two classes of edges in a physical plan that should always be pipelined in an OSRP. The first class of edges, called "bridges," can be pipelined without causing schedulability issues, and must be pipelined in an OSRP. Bridges belong to a more general class of edges, called "clean edges," which are always pipelined in an OSRP. We first define preliminary concepts. An *undirected cycle* C of a physical plan P is a sub-DAG of P such that the underlying graph [3] of C forms a cycle. Figure 11 shows a physical plan with two undirected cycles C_1 and C_2. Intuitively, edges on an undirected cycle can potentially cause a region plan to be unschedulable, and edges not on any undirected cycle will never cause a region plan to be unschedulable.

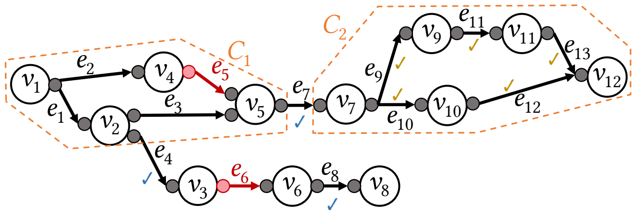


Figure 11. A physical plan P_4 with two undirected cycles C_1 and C_2. An edge with a "✓" mark is a bridge, and an edge with a "✓" mark is a non-bridge clean edge.

Definition 3.6 (Bridge). Given a physical plan P, a non-blocking edge e in P is a *bridge* if e is not in any undirected cycle.

For instance, the non-blocking e_7 in Figure 11 is a bridge because it is not in any of the two undirected cycles C_1 and C_2. Similarly, e_4 and e_8 are also bridges.

Lemma 3.7. *If a region plan U of a physical plan materializes a bridge e and is schedulable, then another region plan U' obtained from U by changing e to a pipelined edge is also schedulable.*

[3]The "underlying graph" of a directed graph P is an undirected graph obtained by replacing all directed edges of P with undirected edges.

This lemma leads to the following corollary.

Corollary 3.8. *A bridge in a physical plan is always pipelined in an optimal schedulable region plan.*

For example, for the physical plan P_4 in Figure 11, all the bridges e_4, e_7, and e_8 must be pipelined in an OSRP. A reason why a bridge is always pipelined in an OSRP is that it is not in the same undirected cycle as another blocking edge. For example, in Figure 11, e_1, e_2, and e_3 are in the same undirected cycle C_1 as a blocking edge e_5. Edge e_5 has to be materialized in a region plan. Pipelining e_1, e_2, and e_3 together causes a region plan to be unschedulable because doing so puts v_1, v_2, v_4, and v_5 in the same region, and then e_5 causes a cycle in the region graph. On the other hand, pipelining all the edges in C_2 does not cause this issue, because there is no blocking edge in this undirected cycle, and all these edges have to be pipelined in an optimal schedulable region plan. Based on this intuition, we formally define the notion of a *clean edge*.

Definition 3.9 (Clean Edge). Given a physical plan P, a non-blocking edge e in P is *clean* if e is not in the same undirected cycle of a blocking edge.

A bridge is always a clean edge, and some clean edges are not bridges. For example, in P_4 (Figure 11), e_9 is not a bridge because it is part of an undirected cycle C_2. The edge is clean since C_2, the only undirected cycle that contains e_9, does not have any blocking edge. Similarly, e_{10}, e_{11}, e_{12}, and e_{13} are also clean edges.

Lemma 3.10. *All the clean edges in an optimal schedulable region plan of a physical plan are pipelined.*

For instance, in P_4 (Figure 11), besides the bridges, the non-bridge clean edges $e_9, \ldots, e_{13}$ must also be pipelined in an OSRP.

4 Top-down Search Framework

In this section, we present a search framework that explores the region-plan plane to find an OSRP and accordingly compute an optimal execution order. We develop techniques to improve its performance, including some based on the properties in Section 3. We continue using the cost function based on the total materialization size (denoted as $Cost_m$).

4.1 The Search Algorithm

Algorithm 1 presents the top-down search framework. We use an example shown in Figure 12 to explain the algorithm. Each region plan with an associated cost in the search space is a *state* in the search process. Given a physical plan P_2, the search space has 16 states $U_0, \ldots, U_{15}$. The search starts from the *seed state* U_0 (line 1), in which all the edges are materialized. The *goal state* is an OSRP U_{13}. Line 2 initializes the optimal state U^* using U_0. Line 3 initializes the search *frontier*, denoted as $\mathcal{F}$, which is the set of known but unexplored states. The *visited-state set* $\mathcal{E}$ includes all the states that have been visited during the expansion till now.

Each edge between two states is a transition. For the current state we expand $\mathcal{F}$ by including the state's neighbors. Additionally, $\mathcal{E}$ is used to avoid repeated additions of the same state into $\mathcal{F}$ (lines 12-15). Note this can save computation but increases the space complexity. We repeat the process of exploring the schedulability and cost of a state from $\mathcal{F}$ and including new neighbor states in $\mathcal{F}$ until $\mathcal{F}$ is empty. An unschedulable state has an infinite cost, and only schedulable states are used to update the optimum U^*. We give U^* to the Scheduler to generate execution orders for U^* and choose an optimal one.

```
Algorithm 1: Top-down search for an optimal execution order
Input  : $P = (V, E)$: a physical plan
         $Cost_m$: a cost function (total materialization size)
Output : $S$: an execution order for $P$ with a minimum cost
1  $U_0 \leftarrow$ the seed state where all edges are materialized ;
2  $U^* \leftarrow U_0$ ; // Initialize the optimum
3  $\mathcal{F} \leftarrow \{U_0\}$ ; // Initialize the frontier
4  $\mathcal{E} \leftarrow \{U_0\}$ ; // Initialize the set of visited states
5  while $\mathcal{F} \neq \emptyset$ do // Stop when all states are explored
6      Remove one state $U_i = (P, \alpha_{U_i}, \beta_{U_i})$ from $\mathcal{F}$ ;
7      if $Cost_m(U_i) < Cost_m(U^*)$ then
8          $U^* \leftarrow U_i$ ; // Update the optimum
9      end
10     foreach $e \in \beta_{U_i}$ not corresponding to a blocking edge in $P$ do // Use non-blocking materialized
        edges for frontier expansion
11         $U_i' \leftarrow (P, \alpha_{U_i} \cup \{e\}, \beta_{U_i} - \{e\})$ ; // Transform the current state into a neighbor state
12         if $U_i' \notin \mathcal{E}$ then // Ensure visiting each state only once
13             Add $U_i'$ to $\mathcal{F}$;
14             Add $U_i'$ to $\mathcal{E}$;
15         end
16     end
17 end
18 $S = generateExecutionOrder(U^*)$ // Choose the best execution order for $U^*$ according to the Scheduler
19 return $S$
```

Completeness and complexity. The search algorithm is complete since its expansion process includes all possible region plans. For a physical plan $P = (V, E)$, it can have at most $2^{|E|}$ region plans. For each state, line 7 takes $O(|V| + |E|)$ time to check schedulability. Lines 10-16 explore at most $|E|$ neighbor states, each taking O(1) time. The time complexity of each iteration is $O(|V| + |E|)$. Hence the total complexity of the algorithm is $O((|V| + 2|E|) \cdot 2^{|E|})$.

Greedy search. The search algorithm is using an exhaustive-search strategy to explore all the execution orders. If the search space is large and the system has limited search time, we can slightly modify the algorithm to perform a greedy search. In particular, we modify lines 10-16 to include only one neighbor state that has the lowest cost among all the neighbors in the frontier.

Next we present several improvement techniques for the search framework.

4.2 Technique: Stopping Exploring Beyond Hopeless States

The search process cannot stop at an unschedulable state. That is, if U_i is unschedulable in line 7, we cannot skip lines 10-16, because some unschedulable states can still lead to schedulable ones.

Next, we show that there are states that will never lead to a schedulable state, and we can stop the search at such states.

We first define a basic concept. A *blocking region edge* in a region graph is an edge that corresponds to a blocking edge in the physical plan. For example, Figure 13 shows the region graph of U_6 in Figure 12, where e_4 is a blocking region edge as it is blocking in P_2.

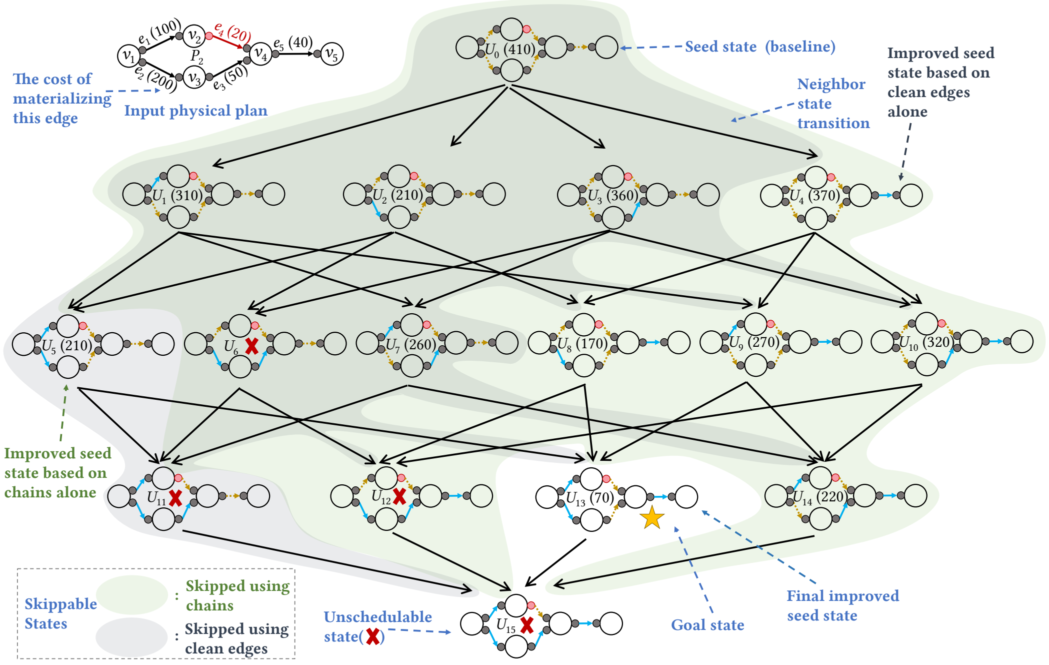


Figure 12. The search process of a simple workflow when running the top-down search Algorithm 1.

Definition 4.1 (Hopeless State). Given a physical plan P, a *hopeless state* is an unschedulable region plan U for P whose region graph G_U has a cycle that contains a blocking region edge.

In Figure 12, U_6 is hopeless because its region graph (shown in Figure 13) has a cycle that contains a blocking region edge e_4.

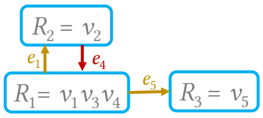


Figure 13. The region graph of U_6 with a blocking region edge in red.

We can show that during the top-down search, U_6 will never lead to a schedulable state.

Lemma 4.2. *A hopeless state cannot be transformed into a schedulable state by changing its materialized edges to pipelined edges.*

To check if a state is hopeless, we can check if there is a blocking edge on a cycle when we detect that the current state is unschedulable (line 7). When there are few blocking edges, this additional step does not incur much overhead. Once a state is determined to be hopeless, we can skip the expansion of its neighbor states (lines 10-16). For instance, during the search in Figure 12, when we explore U_6 and find it to be hopeless, we do not need to add its neighbors U_{11} and U_{12} to the frontier. Furthermore, the states below a hopeless state are also hopeless and should also be excluded. We can do this pruning by using a memoization set to save all the known hopeless states. For example, U_6 will be added to this memoization set. When we expand the neighbor states of U_{13} and see U_{15} is a descendent of U_6, we do not need to add U_{15} to the frontier.

4.3 Technique: Generating a Better Seed State Using Properties of OSRPs

Next, we discuss how to reduce the search space by generating a better seed state based on the properties of OSRPs as described in Section 3. We first discuss how to use chain properties. If we know an edge must be pipelined in an OSRP, we should also pipeline it in the seed state. For instance, consider the physical plan P_2 in Figure 12. According to Lemma 3.3, the edge e_1 has to be pipelined in an OSRP because the edge is on a maximal chain $v_1 \longrightarrow v_2 \longrightarrow v_4$ with a blocking edge e_4. Then e_1 is pipelined in the improved seed state. According to Lemma 3.4, non-blocking edges e_2 and e_3 are on the maximal chain $v_1 \longrightarrow v_3 \longrightarrow v_4$, so they should not both be materialized. Furthermore, as e_3 has the minimal cost on this chain, based on Corollary 3.5, we materialize e_3 and pipeline e_2 in the improved seed state. Finally, the non-blocking edge e_5 is the only one on the maximal chain $v_4 \longrightarrow v_5$, so it still needs to be materialized in the seed. In the end, the improved seed state is U_5, from which we can start the search directly and still find the OSRP U_{13}. This technique helps us reduce the number of states explored from 16 to 4.

We can also use properties of clean edges to improve the seed state. We can use Lemma 3.10 to generate an even better seed state where each clean edge is pipelined. For example, in Figure 12, the non-blocking edge e_5 is clean because it is not in any undirected cycle. Based on the lemma, we can pipeline e_5 and start the search directly from U_4. By applying this technique alone, we can skip the eight states annotated with a yellow background. By combining the techniques using chains and clean edges, we improve the seed state to be U_{13}, leaving only one state in the frontier (U_{15}).

5 Discussion and Generalization

In this section, we present another framework that does the search in the bottom-up direction, discuss how to do cost estimations in Pasta, and extend the results to general cost functions.

5.1 Bottom-up Search Framework

We present another search framework that uses the "bottom-up" direction to explore the search space. It uses the seed state corresponding to the region plan where all the non-blocking edges are pipelined. This state has the minimal total sizes of materialization, but could be unschedulable. In the example in Figure 12, the seed is the bottom state U_{15}. Beginning from this state, we explore the search space by gradually adding more materialized edges to the region plan. That is, each neighbor-state transition changes a pipelined edge to materialized. The bottom-up framework has the same completeness guarantee and time complexity as the top-down search framework, and can also be adapted to perform a greedy search instead of an exhaustive search by adding only one neighbor state with the lowest cost to the frontier. Next, we present several techniques to optimize the performance of the search.

Technique: Stopping exploring beyond schedulable states. When a state is already schedulable, changing its pipelined edges to materialized can only increase the cost, and will not lead to an optimal schedulable region plan. Therefore, during the search process, whenever the current state is already schedulable, there is no need to include its neighbor states in the frontier.

Technique: Pruning using properties of OSRPs. The properties of chains and clean edges can also be utilized to optimize the bottom-up search. The main idea is to avoid transitioning into a state that violates the properties of chains and clean edges. In particular, because a state transition can only change an edge from pipelined to materialized, if we encounter a state that materializes an edge that must be pipelined in an OSRP, then we should stop exploration beyond this state. To use properties of chains, we change a pipelined edge e of a region plan to materialized if e satisfies the following conditions. (1) According to Lemma 3.3, e should not be in a chain that contains a blocking edge. Furthermore, according to Lemma 3.4, if e is in a chain without a blocking edge, and the chain already has a materialized edge, then e should not be changed. (2) According to Corollary 3.5, if e is in a chain with only pipelined edges, then e needs to have the lowest cost among these edges. Moreover, as a maximal chain covers the pruning power of its sub-chains, we could pre-compute all the maximal chains in the physical plan and use only maximal chains to do these checks to improve the efficiency of applying these two cases. Similarly, clean edges can also be used to do pruning during the search. In particular, according to Lemma 3.10, we can exclude any region plan with a clean edge materialized, as a clean edge should not be changed from pipelined to materialized.

Table 3 summarizes the applicability of each technique in the two search frameworks.

Table 3. Applicability of techniques on the two search frameworks.

Property	**Description regarding an OSRP**	**Used in Top-down Search**	**Used in Bottom-up Search**
Property A	Pipelines all non-blocking edges on a chain with a blocking edge.	Generate a better seed	Prune states
Property B	Has at most one materialized non-blocking edge on a chain without a blocking edge.	Not applicable	Prune states
Property C	Pipelines an edge on a chain without a blocking edge and containing a lower-cost edge.	Generate a better seed	Prune states
Property D	Pipelines a clean edge.	Generate a better seed	Prune states
Property E	Is unreachable from a hopeless region plan by changing materialized edges to pipelined.	Prune states	Not applicable
Property F	Is unreachable from a schedulable region plan by changing pipelined edges to materialized.	Not applicable	Prune states

5.2 Obtaining Cost Information

There are various ways to obtain costs of execution orders. One way is to use a two-phase execution method. In the first phase, we use heuristics to generate an execution order (e.g., one with all edges materialized), and run it on a small subset of the source data. After this execution, we collect information about the materialization size of each edge on the sample data. In the second phase, we use the information to estimate the cost of each edge. Another way is to use the iterative nature of data analytics tasks, i.e., one workflow may often be executed multiple times [6]. Thus we can utilize previous executions for cost estimation. The first execution can be done using any existing methods. For example, Hyracks [3] uses a heuristic where both input edges to a two-input operator are materialized, or we can use the aforementioned execution method. During each subsequent execution, Pasta has information about the statistics of each edge, and uses it to estimate costs of an execution order. In general, having high-quality cost information is important. Many existing techniques [30] have been proposed to tackle this issue. We can also first utilize the two aforementioned methods to obtain cost information that is "good enough" to empower Pasta initially, and gradually improve the statistics to perform re-optimizations when we find them to be inaccurate.

5.3 General Cost Functions

So far we have used a simple but commonly used function that measures the total materialization size of an execution order. Next, we consider general cost functions and extend the results. For a general cost function C, we can still use both search frameworks to perform a search to find an optimal execution order. In the resulting OSRPs, Property E (Section 4.2) still holds, as a hopeless state cannot lead to schedulable states by having more pipelined edges. Other OSRP properties are no longer valid. Next, we present two conditions for cost functions:

- Condition I: given a region plan U, if another region plan U' is derived from U by changing a materialized edge in U to a pipelined edge, then either U' is unschedulable or $Cost(U') \leq Cost(U)$.
- Condition II: the cost of an execution order S for a region plan U is the summation of a constant cost of each edge in the region plan of this execution order, i.e., $Cost(S) = \sum_{e \in E_U} Cost(e)$, where $Cost(e)$ is the cost of an edge e.

Figure 14 shows the relationships between these conditions and the OSRP properties. An example cost function that satisfies Condition I but not Condition II is based on the total wall-clock time of an execution order.

Cost Function B (total wall-clock time). The cost of the total wall-clock time of an execution order S for a region plan U is the summation of the execution time of the longest-running operator in each region, i.e., $Cost_t(S) = \sum_{R \in V_{G_U}} \max_{v \in R} Time(v)$, where $Time(v)$ is the execution time of an operator v.

This cost model assumes that given a region plan, the Scheduler only considers the execution order that corresponds to a total order on the regions of a region plan. This is a coarse model on the total wall-clock time, and it is useful for many workflows where the completion time of a region is dominated by long-running operators, e.g., ML-inference operators. For this cost function, the OSRP properties A, B, D, and F still hold, since they only need the cost function to penalize having more materializations. Property C does not hold because it assumes an individual cost on each edge. Finally, when a cost function satisfies both conditions, Property C still holds. Notice that the cost function of the total materialization size belongs to this category.

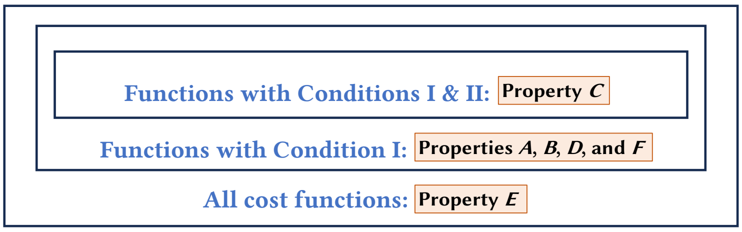

Functions with Conditions I & II:
Property C
Functions with Condition I:
Properties A, B, D, and F
All cost functions:
Property E

Figure 14. Relationships between conditions of cost functions and properties of OSRPs.

6 Experiments

In this section we report the results of a thorough evaluation of Pasta on real-world workflows.

6.1 Settings

Workflow Engine. We conducted experiments on the Texera system [26], an open-source data workflow platform based on the Amber engine [16]. Texera is developed in Scala and supports push-based execution. It supports pipelined execution of physical plan DAGs and includes a flow-control mechanism to apply backpressure between operators. A workflow in Texera is constructed by a user on a GUI-based interface via drag-and-drop operations and processes bounded input data. A workflow is submitted to Texera as a logical plan DAG that is compiled to a physical plan, where a logical operator (e.g., HashJoin) may be expanded to multiple physical operators (e.g., Build followed by Probe). Physical operators are annotated with information about blocking ports. Texera also supports data-parallel execution of a physical operator via partitioning. Currently Texera does not support control operators [9], e.g., for-loops.

Pasta Implementation. We implemented Pasta in Texera. We implemented both the top-down search (Section 4) and bottom-up search (Section 5.1), along with the improvement techniques, including Chains (CHN), Clean Edges (CLE), and Early Stop (ESP). Each search framework included global search and greedy search. Given a region plan, the Scheduler module of Pasta generated a final execution order by using a topological sort of the regions, and a workflow was executed region-by-region.

Other Scheduling Methods. We implemented several non-cost-based methods for generating execution orders in Texera for comparison purposes. "Bottom-up Seed" (BUS) is the method of generating a region plan using the seed of the bottom-up search framework, which may be unschedulable. "Top-down Seed" (TDS) refers to generating an execution order using the seed of the top-down search framework with no improvement techniques, which guarantees an execution order but also prevents pipelining. Finally, `Baseline` is a method corresponding to the existing heuristics used by systems such as Hyracks, Flink, and Texera. `Baseline` starts from the seed of the top-down search process, traverses the edges of the region plan following a topological order of the operators, and changes a non-blocking materialized edge to pipelined (which leads to a new region plan) if and only if the new region plan is schedulable. As a result, the final region plan is also guaranteed to be schedulable. For all these methods, we also generated an execution order from a region plan by using a total order of the regions.

Optimization Goals and Machines. We used two optimization goals to evaluate Pasta's performance. The first was minimizing the total sizes of materialization (*Mat-Sizes*) of a workflow. As we will show in Section 6.2, the availability of numerous physical plans with cost information when using this goal allowed us to evaluate Pasta at a large scale. The experiments for this optimization goal were conducted on a machine running the Ubuntu 22.04 operating system, with 64 cores, 128 GB of memory, and an 8 TB hard disk. The second was reducing the wall-clock time of executing a workflow (*Clock-Time*). We used this goal to show the generality of Pasta as well as its performance in an end-to-end workflow-execution setting. The experiments of this optimization goal were conducted on a machine running MacOS Sonoma 14.5, with 10 cores, 32 GB of memory, and a 1 TB SSD. Each experiment in the evaluations was executed three times, and the averages were reported.

6.2 Collecting Real Workflows

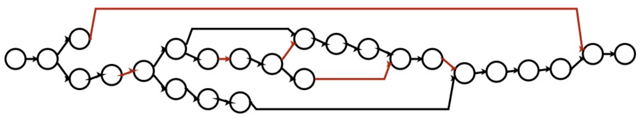
We used 6,148 real-world workflows from the KNIME Community Hub [14] for analysis. The KNIME engine does not support pipelined execution. Instead, it executes a physical plan (DAG) operator by operator and materializes intermediate results. Since Pasta requires pipelined execution, it was not feasible to use the KNIME engine to evaluate Pasta. We converted those workflows to a format supported by Texera. During a conversion, the DAG structure was preserved, and each KNIME operator was mapped to a Texera operator. All the ports and edges were preserved. Around 20% of the workflows contained control operators, which were treated as normal data-processing operators during the conversion. For KNIME operators that were not available in Texera, we replaced them with non-executable Dummy operators. For each output port of a Dummy operator, we annotated whether it was blocking using the following criteria. (1) If a port was to produce a single object (e.g., an ML model) instead of a data table, it was blocking. (All KNIME operator ports were already marked with this information.) (2) If a port was part of a control operator (e.g., Loop Start), it was blocking. (3) For the remaining cases, we inferred whether a port was blocking based on the official operator description from KNIME. Each workflow contained one or more weakly connected components, and each component corresponded to a physical plan DAG ("plan" for short in the rest of this section). As each plan could be scheduled separately, we used the largest plan (in terms of number of operators) of each workflow for the analysis and experiments. We derived subsets of those workflows for specific experiments, summarized in Table 4.

Table 4. Overview of workflow (WF) sets used in evaluations.

WF Set	WF #	Source	Use
6K-Set	6,148	Migrated to Texera from KNIME Workflows	Analysis (Section 6.3)
MS-Set	849	Subset of 6K-Set	*Mat-Sizes* Evaluations (Section 6.4 & Section 6.5)
CT-Set	2	Created as executable workflows in Texera	*Clock-Time* Evaluations (Section 6.6)

Workflows used for *Mat-Sizes* Evaluations. The *Mat-Sizes* experiments were performed on MS-Set, which was a subset of 6K-Set. We will describe details about MS-Set in Section 6.3. We used the cost function in Section 2.5, and used the materialization information of the original KNIME workflows as the materialization costs.

Workflows Used for *Clock-Time* Evaluations. We manually created two executable workflows in Texera, namely WF$_1$ and WF$_2$, based on two KNIME workflows in 6K-Set. Their physical plans are shown in Figure 15. WF$_1$ contained 25 operators and 28 edges to process a dataset of 100K email addresses and used a combination of rule-based methods and ML models to identify fraudulent addresses. WF$_2$ processed a dataset of 5,000 movies, each containing attributes like titles, genres, synopses, etc. The objective was to leverage pre-trained models for text summarization and sentiment analysis on textual attributes, and to generate one-hot encodings for categorical attributes. Its physical plan contained 35 operators and 39 edges.


(a) Workflow 1 (WF$_1$)

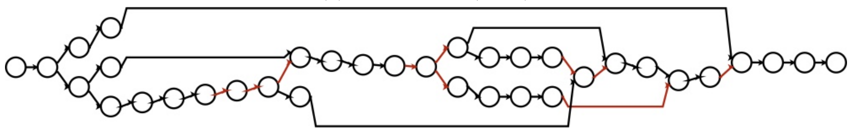

(b) Workflow 2 (WF$_2$)

Figure 15. Physical plans evaluated on the *Clock-Time* goal.

6.3 Complexity of Scheduling Workflows

To understand the complexity of finding optimal execution orders on these workflows, we analyzed the plans in 6K-Set.

Structural Complexity of Physical Plans. We measured the plans in terms of the complexity of their DAG structures. We first counted the scale of the plans, measured by the number of operators and number of edges in a DAG. Figure 16 shows the histogram of the number of operators and number of edges in each DAG among the 6K plans. More than half of the DAGs had at least 10 operators; more than 20% of the DAGs contained at least 20 operators; and over 6% of the DAGs had at least 50 operators. The distribution for the edge numbers was similar. Moreover, 83 DAGs contained 100+ operators, 99 DAGs had 100+ edges, and the largest plan included 465 operators and 742 edges. These statistics show that the workflow DAGs had a large scale. We also observed that the number of edges in many of the plans was higher than the number of operators. This trend was also reflected in Figure 16. The decrease in frequencies of plans with a larger edge count was slower than that of the operator count. One reason was the high frequency of non-tree-DAGs, which typically have more edges than operators. We measured the proportion of non-tree DAGs in 6K-Set. Out of all the 6K plans, 3,208 (52.2%) of them were non-tree DAGs, and this ratio was higher (77.7%) among plans with more than 10 operators. Scheduling tree-based plans for pipelined execution tends to be easier compared to scheduling non-tree plans. For instance, according to Lemma 3.8, all non-blocking edges in a tree should be pipelined since they are all bridges. Therefore, for the cost function of *Mat-Sizes* on a tree-based physical plan, using the BUS method is sufficient.

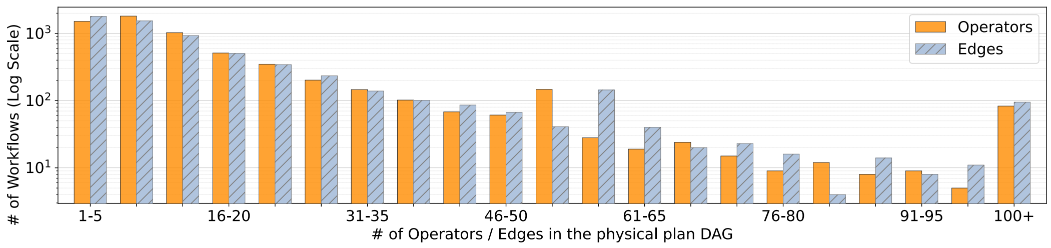


Figure 16. Distribution of operator numbers and edge numbers in physical plan DAGs of the 6K workflows.

Schedulability of BUS Region Plans. We next measured the proportions of plans that needed non-simple methods to enable pipelined execution. We considered BUS to be a simple method. The reason is that according to Property F (Section 5.3), for the *Mat-Sizes* and *Clock-Time* cost functions, if the seed of the bottom-up search is schedulable, it can be directly used to generate an optimal execution order. Otherwise, either `Baseline` or Pasta is needed to generate an execution order. For each physical plan in 6K-Set, we used BUS to generate a region plan and tested its schedulability. In total, there were 1,259 physical plans (20.4% of all the physical plans, and 39.2% of all the non-tree DAGs) that had an unschedulable BUS region plan. Figure 17 shows the fraction of unschedulable BUS region plans with varying scales of physical plans for 6K-Set and for the subset where physical plans were non-tree DAGs. In general, physical plans with larger scales were more likely to have an unschedulable BUS region plan. Moreover, when a physical plan (including tree-structured plans) had more than 10 operators, this ratio was consistently above 20%, and the cut-off increased to 40% when the scale was above 30 operators. This ratio among the non-tree DAGs was higher, with the cut-off being 20% among physical plans of all scales. These statistics show that it is challenging to schedule many workflows for pipelined execution.

Physical Plans in MS-Set. We used 849 of the 6K plans for *Mat-Sizes* evaluations. When the BUS method produced an execution order, there was no need to use other methods. Thus, only the 1,259 physical plans on which BUS produced an unschedulable region plan needed evaluations. Note all these plans were non-tree DAGs. Furthermore, out of the 1,259 physical plans, 849 (67.4%) of them had the cost information available because they were executed in KNIME before, which resulted in MS-Set. Plans of all sizes were preserved in MS-Set, from smaller plans (operator count < 15) to very large plans (operator count ≥ 100, a total of 41 plans).

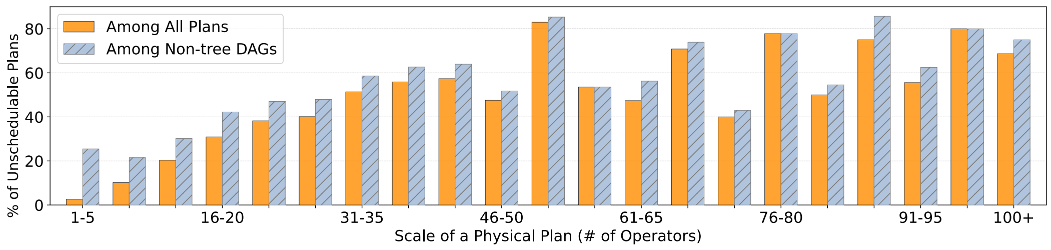


Figure 17. Ratio of physical plans whose BUS region plan was not schedulable.

6.4 Effectiveness of Pasta on *Mat-Sizes*

Next, we evaluated the effect of using Pasta on improving the quality of a generated execution order. We used MS-Set in the evaluation. To measure the materialization cost, we did not need to execute the workflows, and we conducted the analysis on the generated execution orders for each workflow. Given each of these 849 plans as an input, we used different methods to generate execution orders, including Pasta, TDS, and Baseline. We evaluated four configurations of Pasta (ETD, GTD, EBU, and GBU). We mainly present Pasta's results when using the top-down search direction (ETD and GTD) since the bottom-up search direction (EBU and GBU) had similar results.

We enforced a time limit of 300 seconds for each individual run of execution-order generation. For Pasta methods, if this time limit was reached, we used the best region plan found so far to generate a final execution order. For each physical plan, if ETD and GTD produced different execution orders, we used the better one as the result of Pasta. We compared the cost of execution orders generated by Pasta with those from Baseline and TDS. Note TDS represented another simple method that materialized all the edges, which had the highest possible cost. As the materialization sizes varied greatly across different physical plans, we used relative costs (Pasta or Baseline relative to TDS) for comparisons. For each physical plan, we measured the materialization size S_m of the execution order produced by TDS, as well as the size S_f of the execution order produced by Pasta or Baseline. We adjusted S_m and S_f by subtracting the total sizes of blocking edges S_b in the plan because these edges had to be materialized in each method. We used $(S_f - S_b)/(S_m - S_b)$ to represent the relative cost of an execution order generated by Pasta or Baseline.

Pasta vs. Other Methods. Pasta produced the best execution order among the three methods on 847 of the 849 evaluated plans. 33 (3.9%) out of these 847 execution orders were not produced by ETD because it was unable to finish within 300 seconds on these plans. Instead, they were produced by GTD. Pasta produced a worse execution order than that from Baseline only for the two large plans on which even GTD was unable to finish execution within 300 seconds. On 559 (65.8%) plans the cost of execution orders generated by Pasta was the same as that of Baseline because the two methods produced the same execution order. On the remaining 288 (33.9%) plans, the execution order produced by Pasta had a lower cost than that of Baseline.

To further study the extent to which Pasta could reduce costs, we calculated the ratio of cost reduction when using Pasta compared to Baseline on each of these 288 workflows. Given the relative cost C_b of an execution order from Baseline and the relative cost C_p of that from Pasta, this ratio was $(C_b - C_p)/C_b$. We show these ratios as well as the relative costs of the two methods on these 288 instances in Figure 18. On less than a quarter of these workflows, Pasta improved the costs by less than 10%; on more than half of these instances, Pasta's execution order could reduce over 30% of the costs from Baseline; on around 1/3 of them, Pasta's execution order was 50% better than that of Baseline; in some cases Pasta was even able to reduce more than 99% of the cost compared to Baseline. The relative costs of Pasta's execution orders were almost consistently below 50%.

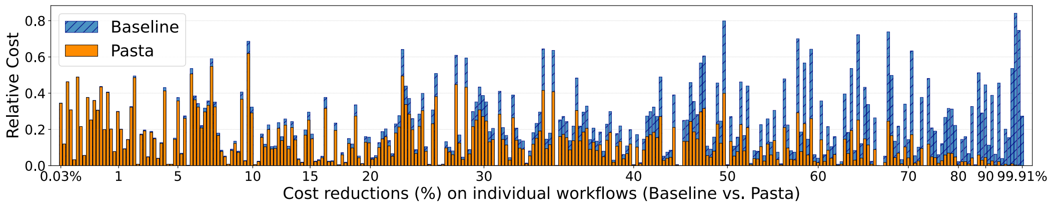


Figure 18. Materialization costs of execution orders generated by Pasta and Baseline on all the workflows where the two methods produced different results. Each bar represents a workflow. The relative costs of execution orders from the two methods are overlapped, and the visible bars for Baseline indicate reduced relative costs when using Pasta over Baseline. The workflows are ordered by the cost-reduction ratio of using Pasta compared to Baseline.

Greedy Search vs. Exhaustive Search. In the top-down framework, Pasta offers both an exhaustive search method and a greedy search method. We compared the relative costs of ETD and GTD on the 812 instances that both methods finished execution on. GTD produced the same execution order as ETD (i.e., optimal execution order) on 703 (86.6% out of 812) of the instances. This meant that the faster method GTD was often as effective as the slower method ETD, so using GTD for quickly generating an execution order would be beneficial. For recurring and costly jobs, if optimal scheduling is desired, we can run an exhaustive search in the background for further improvements. Figure 19 shows the cost-comparison results on the remaining 109 (13.4%) instances where the execution order produced by ETD had a lower cost. For over half of them, the execution order from ETD reduced more than 30% of the costs compared to the one from GTD. On several instances, the optimal execution order of ETD reduced over 99% of the cost from that of GTD. These results show that, compared to the greedy search, using an exhaustive search for an optimal execution order can further reduce costs.

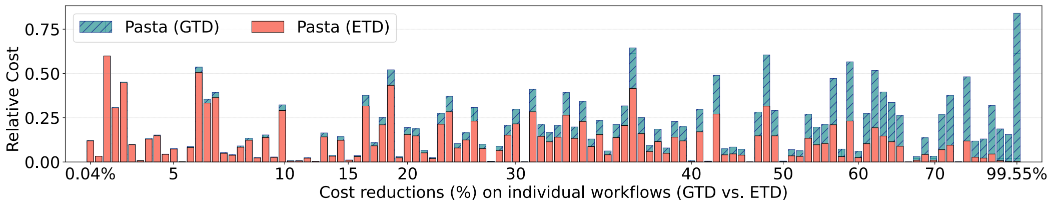


Figure 19. Materialization costs of execution orders generated by ETD and GTD on all the workflows where the two methods produced different results. Similar to Figure 18, the relative costs of execution orders from the two methods are overlapped, and the workflows are ranked by the cost reduction ratio of using ETD over GTD.

6.5 Efficiency of Pasta on *Mat-Sizes*

Next, we evaluated the efficiency of Pasta. We mainly present Pasta's results when using the top-down search direction. We first present the overall results, then show detailed evaluations on two aspects: (1) *Input complexity*: How did the scale of input plans affect Pasta's performance? (2) *Breakdown of improvement techniques*: How effective were the techniques (CHN, CLE, and ESP)?

Overall Efficiency. We first considered the efficiency of the exhaustive-search method. Given the time constraint of 300 seconds, ETD finished on 812 (95.6%) of the 849 plans. Out of the 37 plans on which ETD exceeded the time limit, 33 (89.2%) of the plans had over 80 operators and over 66 non-blocking edges. This meant that although ETD needed a longer time to find an optimal execution order for some complicated plans, it could finish within a reasonable amount of time on the vast majority of real-world plans. Moreover, when we decreased the cut-off time to 5 seconds, ETD was able to finish scheduling on 675 (79.5%) of the plans. These statistics showed that ETD was very efficient on most of the plans. Finally, on average the scheduling time of ETD was 13X the time of Baseline, and for 766 (90.2%) of the plans this ratio was lower than 13. This meant that with additional search time, ETD could find optimal execution orders that sometimes were much better than those from Baseline. We next considered the greedy method. Within 300 seconds, GTD finished on 846 (99.6%) of the 849 plans. There were two plans on which GTD exceeded the time limit. One had 302 operators and 375 edges, and another contained 465 operators and 742 edges. When we reduced the cut-off time to 2 seconds, GTD finished on 767 (90.3%) of the plans. On average the time for GTD to finish was 4X that of Baseline. Considering the fact that GTD generated an equal or better execution order compared to Baseline on 844 (99.4%) of the plans, using GTD over Baseline would be a sensible choice.

Impact of Physical Plan Scales. To study the impact of input complexity on the efficiency of the methods, we measured the average scheduling time of each method on different scale-groups of physical plans (measured by the number of operators), as shown in Figure 20. For all the methods, the scheduling time increased with the growing complexity of the input plan. Moreover, although GTD was slower than Baseline, the growth of GTD's scheduling time had a similar shape as that of Baseline, because both methods had a polynomial time-complexity. ETD was faster than GTD for smaller plans because the search space was small and GTD did not have the extra step of comparing costs of different neighbors of each visited state during the search. For larger plans, the time for ETD increased more rapidly and erratically than GTD, because the search space grew exponentially. Nevertheless, the improvement techniques helped ETD maintain its efficiency for most inputs.

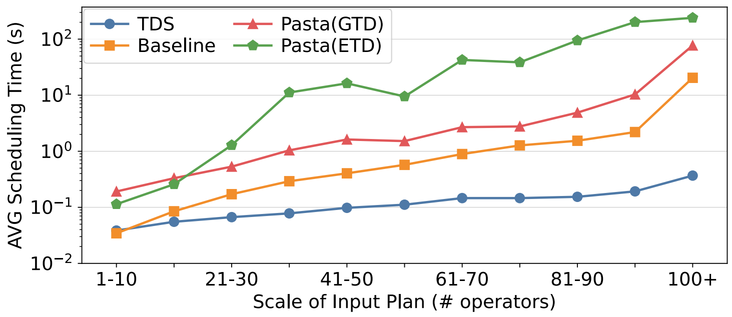


Figure 20. Given a physical plan, the average time it took to generate an execution order for different methods across varying workflow scales.

Effect of Improvement Techniques. Finally, we studied the effect of each improvement technique on Pasta's performance. We first measured their impact on the exhaustive-search method (ETD). For each plan, we used five methods to generate an execution order with a time budget of 30 seconds. These searches included: one without any improvement technique (NOP), one for each technique applied individually (CHN, ESP, and CLE), and one with all techniques applied together (Pasta). We categorized the results based on the scale of the input plans. We then calculated the ratio of workflows on which each method successfully finished the search to identify an optimal execution order within the budget for each group. Figure 21 shows the success ratios on different groups for each method. NOP succeeded on only plans with smaller scales, and was mostly unsuccessful once the scales went above 20. ESP alone increased the ratio moderately for smaller plans. CLE alone greatly increased the success ratio for many plans, and CHN was the most effective. We attribute the high effectiveness of CHN and CLE to the fact that most of the input plans had many chains and clean edges, which enabled the two techniques to reduce the search space significantly. By combining the three techniques, Pasta achieved the best performance.

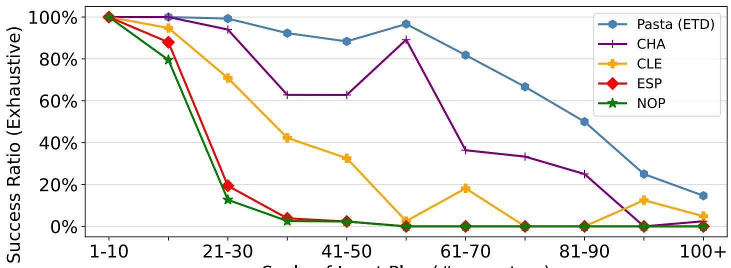


Figure 21. With a cut-off time of 30s, the success ratio of exhaustive searches across workflows using different improvement techniques. NOP refers to the method of not using any improvement technique.

We did similar experiments on the greedy-search method (GTD). Due to its shorter search time, we allocated a smaller time budget of 2 seconds. Figure 22 shows results similar to those in ETD, and the improvements of combining the three techniques were less.

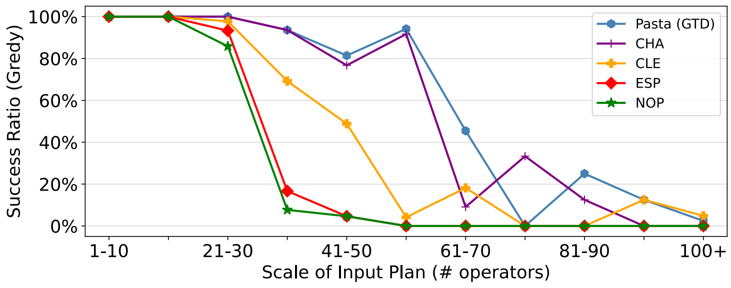


Figure 22. With a cut-off time of 2s, the success ratio of greedy searches across plans using different improvement techniques.

6.6 Evaluations on the *Clock-Time* Goal

So far, we presented evaluations of Pasta using the *Mat-Sizes* cost function. Next, we considered another optimization goal, namely *Clock-Time*. The purposes are (1) to evaluate the generality of Pasta on other cost functions; and (2) to measure the end-to-end performance of Pasta in Texera.

For both workflows WF_1 and WF_2, we used TDS, `Baseline`, and Pasta (EBU) to generate execution orders. We varied the input size of each workflow and ran multiple executions. For each execution, we recorded the scheduling time to generate the final execution order, the cost of the final execution order, and the end-to-end wall-clock time to finish workflow execution. For each workflow, we executed the workflow by using TDS to obtain the wall-clock time of each operator and used these numbers as the cost for Pasta.

Figure 23 shows WF_1's costs and end-to-end execution time on different sizes of the input data. The execution orders generated by Pasta remained the same for different input sizes. This was because the size of the input data did not significantly affect how the costs differed between operators in the same workflow. The lowest costs of Pasta's execution orders demonstrated its effectiveness in reducing the costs for the *Clock-Time* goal. Moreover, the reduced costs also resulted in reduced end-to-end clock time, and the gap between different methods grew larger with the increase of the input size. In addition, Pasta's scheduling time remained consistent across different input sizes, with an average of 64 ms. This was relatively small compared to the end-to-end execution time, especially for larger input sizes.

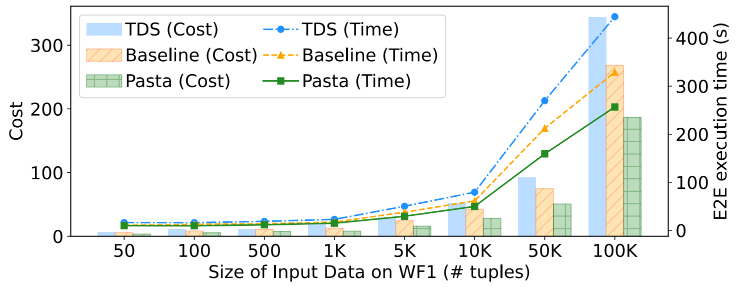


Figure 23. Results of *Clock-Time* evaluations on WF_1.

Figure 24 shows the results for WF_2. Pasta provided greater benefits for this workflow compared to WF_1 due to the higher costs of the operators. The scheduling time of Pasta on this more complicated plan remained stable, with an average of 166 ms. These results showed both the generality of Pasta when using another cost function and its ability to reduce end-to-end workflow execution time with low overhead.

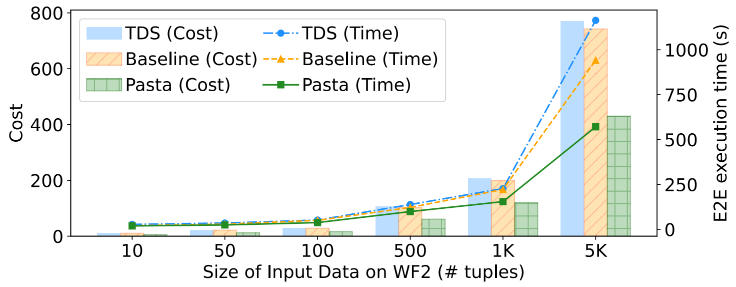


Figure 24. Results of *Clock-Time* evaluations on WF_2.

7 Conclusions

In this paper, we studied the problem of generating an optimal execution order of pipelined execution for a dataflow DAG. We developed a novel cost-based optimizer called Pasta, which considers multiple region plans for a physical plan to generate an optimal execution order. The Pasta optimizer is applicable to many cost functions, and can utilize properties in these cost functions to improve its performance. We conducted a thorough evaluation on real-world workflows and showed the efficiency and efficacy of the proposed techniques.

Acknowledgments

We thank Jiadong Bai, Yunyan Ding, Shengquan Ni, Kun Woo (Chris) Park, Vincent Thai, Zuozhi Wang, and Tianyun Yuan for their help. We also thank our anonymous shepherd and reviewers for their invaluable feedback. This work was supported by NSF award IIS-2107150 and NIH NIDDK award 2U24DK097771-11.

References

[1] Alteryx 2024. AI Analytics Platform - Alteryx, https://www.alteryx.com/.

[2] Apache Flink 2024. Apache Flink® — Stateful Computations over Data Streams | Apache Flink, https://flink.apache.org.

[3] Vinayak R. Borkar, Michael J. Carey, Raman Grover, Nicola Onose, and Rares Vernica. 2011. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan (Eds.). IEEE Computer Society, 1151–1162. https://doi.org/10.1109/ICDE.2011.5767921

[4] Luc Bouganim, Daniela Florescu, and Patrick Valduriez. 1996. Dynamic Load Balancing in Hierarchical Parallel Database Systems. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda (Eds.). Morgan Kaufmann, 436–447. http://www.vldb.org/conf/1996/P436.PDF

[5] Nilesh N. Dalvi, Sumit K. Sanghai, Prasan Roy, and S. Sudarshan. 2001. Pipelining in Multi-Query Optimization. In *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 21-23, 2001, Santa Barbara, California, USA*, Peter Buneman (Ed.). ACM. https://doi.org/10.1145/375551.375561

[6] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Zoi Kaoudi, Tilmann Rabl, and Volker Markl. 2022. Materialization and Reuse Optimizations for Production Data Science Pipelines. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 1962–1976. https://doi.org/10.1145/3514221.3526186

[7] Docker Swarm 2024. Swarm mode | Docker Docs, https://docs.docker.com/engine/swarm/.

[8] Yannis Foufoulas and Alkis Simitsis. 2023. User-Defined Functions in Modern Data Engines. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 3593–3598. https://doi.org/10.1109/ICDE55515.2023.00276

[9] Gábor E. Gévay, Tilmann Rabl, Sebastian Breß, Lorand Madai-Tahy, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2021. Efficient Control Flow in Dataflow Systems: When Ease-of-Use Meets High Performance. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 1428–1439. https://doi.org/10.1109/ICDE51399.2021.00127

[10] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 81–97. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/grandl_graphene

[11] Waqar Hasan and Rajeev Motwani. 1994. Optimization Algorithms for Exploiting the Parallelism-Communication Tradeoff in Pipelined Parallelism. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo (Eds.). Morgan Kaufmann, 36–47. http://www.vldb.org/conf/1994/P036.PDF

[12] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*, David G. Andersen and Sylvia Ratnasamy (Eds.). USENIX Association. https://www.usenix.org/conference/nsdi11/mesos-platform-fine-grained-resource-sharing-data-center

[13] KNIME 2024. Open for Innovation | KNIME, https://www.knime.com/.

[14] KNIME Community Workflows 2024. Workflows | KNIME Community Hub, https://hub.knime.com/search?type=Workflow.

[15] Kubernetes 2024. Kubernetes, https://kubernetes.io/.

[16] Avinash Kumar, Zuozhi Wang, Shengquan Ni, and Chen Li. 2020. Amber: A Debuggable Dataflow System Based on the Actor Model. *Proc. VLDB Endow.* 13, 5 (2020), 740–753. https://doi.org/10.14778/3377369.3377381

[17] V. S. Anil Kumar, Madhav V. Marathe, Srinivasan Parthasarathy, and Aravind Srinivasan. 2009. Scheduling on Unrelated Machines under Tree-Like Precedence Constraints. *Algorithmica* 55, 1 (2009), 205–226. https://doi.org/10.1007/S00453-007-9004-Y

[18] Xiaozhen Liu, Zuozhi Wang, Shengquan Ni, Sadeem Alsudais, Yicong Huang, Avinash Kumar, and Chen Li. 2022. Demonstration of Collaborative and Interactive Workflow-Based Data Analytics in Texera. *Proc. VLDB Endow.* 15, 12 (2022), 3738–3741. https://www.vldb.org/pvldb/vol15/p3738-liu.pdf

[19] Ming-Ling Lo, Ming-Syan Chen, Chinya V. Ravishankar, and Philip S. Yu. 1993. On Optimal Processor Allocation to Support Pipelined Hash Joins. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*, Peter Buneman and Sushil Jajodia (Eds.). ACM Press, 69–78. https://doi.org/10.1145/170035.170053

[20] Pipelined Regions in Apache Flink 2020. Improvements in task scheduling for batch workloads in Apache Flink 1.12, https://flink.apache.org/2020/12/02/improvements-in-task-scheduling-for-batch-workloads-in-apache-flink-1.12/.

[21] Raghu Ramakrishnan and Johannes Gehrke. 2002. *Database Management Systems*. WCB/McGraw-Hill.

[22] RapidMiner 2024. Data Analytics and AI Platform | Altair RapidMiner , https://rapidminer.com/.

[23] Vladislav Shkapenyuk, Ryan Williams, Stavros Harizopoulos, and Anastassia Ailamaki. 2005. Deadlock resolution in pipelined query graphs. Carnegie Mellon University Technical Report.

[24] Evan R. Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J. Franklin, and Benjamin Recht. 2017. KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. IEEE Computer Society, 535–546. https://doi.org/10.1109/ICDE.2017.109

[25] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin C. Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: yet another resource negotiator. In *ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013*, Guy M. Lohman (Ed.). ACM, 5:1–5:16. https://doi.org/10.1145/2523616.2523633

[26] Zuozhi Wang, Yicong Huang, Shengquan Ni, Avinash Kumar, Sadeem Alsudais, Xiaozhen Liu, Xinyuan Lin, Yunyan Ding, and Chen Li. 2024. Texera: A System for Collaborative and Interactive Data Analytics Using Workflows. *Proc. VLDB Endow.* 17, 11 (2024), 3580–3588. https://www.vldb.org/pvldb/vol17/p3580-wang.pdf

[27] Zuozhi Wang and Chen Li. 2023. Building a Collaborative Data Analytics System: Opportunities and Challenges. *Proc. VLDB Endow.* 16, 12 (2023), 3898–3901. https://doi.org/10.14778/3611540.3611580

[28] Yinggen Xu, Liu Liu, and Zhijun Ding. 2020. DAG-Aware Joint Task Scheduling and Cache Management in Spark Clusters. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, May 18-22, 2020*. IEEE, 378–387. https://doi.org/10.1109/IPDPS47924.2020.00047

[29] Zhihui Yang, Zuozhi Wang, Yicong Huang, Yao Lu, Chen Li, and X. Sean Wang. 2022. Optimizing Machine Learning Inference Queries with Correlative Proxy Models. *Proc. VLDB Endow.* 15, 10 (2022), 2032–2044. https://www.vldb.org/pvldb/vol15/p2032-yang.pdf

[30] Shaoyi Yin, Abdelkader Hameurlain, and Franck Morvan. 2015. Robust Query Optimization Methods With Respect to Estimation Errors: A Survey. *SIGMOD Rec.* 44, 3 (2015), 25–36. https://doi.org/10.1145/2854006.2854012

[31] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*. 15–28.

Received April 2024; revised July 2024; accepted August 2024