HYPPO: Using Equivalences to Optimize Pipelines in Exploratory Machine Learning

Antonios Kontaxakis
ULB, Brussels, Belgium
UPC, Barcelona, Spain
antonios.kontaxakis@ulb.be

Dimitris Sacharidis
Université Libre de Bruxelles
Brussels, Belgium
dimitris.sacharidis@ulb.be

Alkis Simitsis
Athena Research Center
Athena, Greece
alkis@athenarc.gr

Alberto Abelló
Universitat Politècnica de Catalunya
Barcelona, Spain
alberto.abello@upc.edu

Sergi Nadal
Universitat Politècnica de Catalunya
Barcelona, Spain
sergi.nadal@upc.edu

***Abstract*—We present HYPPO, a novel system to optimize pipelines encountered in exploratory machine learning. HYPPO exploits alternative computational paths of artifacts from past executions to derive better execution plans while reusing materialized artifacts. Adding alternative computations introduces new challenges for exploratory machine learning regarding workload representation, system architecture, and optimal execution plan generation. To this end, we present a novel workload representation based on directed hypergraphs, and we formulate the problem of discovering the optimal execution plan as a search problem over directed hypergraphs and that of selecting artifacts to materialize as an optimization problem. A thorough experimental evaluation shows that HYPPO results in plans that are typically one order (up to two orders) of magnitude faster and cheaper than the non-optimized pipeline and considerably (up to one order of magnitude) faster and cheaper than plans generated by the state of the art when materializing artifacts is possible. Lastly, our evaluation reveals that HYPPO reduces the cost by 3–4× even when materialization cannot be exploited.**

I. Introduction

Exploratory Machine Learning. Developing Machine Learning (ML) solutions involves an exploration phase with multiple ML pipelines, each having different combinations of data preprocessing, feature selection, model selection, or evaluation metrics. Exploratory ML (EML) describes the iterative process of trial and error, where an ML pipeline is revised and refined until a satisfactory level of performance is achieved.

Exploratory search in ML is typically bound by time or cost concerns. Recent studies argue that the main impediment to the adoption of ML in many organizations is the high costs incurred by running ML pipelines [1], [2]. One direction to reduce such costs is to employ AutoML [3] techniques to reduce the *number* of pipelines that need to be investigated. A different, orthogonal approach is to reduce the *cost* of pipelines by applying *optimization techniques*, such as reuse and materialization [4], [5], [6], [7]. The latter approach seems very promising in exploratory scenarios, where an ML engineer executes pipelines that construct or require artifacts computed in past iterations, uncovering *within-experiment* reuse opportunities. Moreover, in large organizations, multiple data scientists work on the same data and perform similar ML tasks, thus presenting *across-experiments* reuse opportunities.

Optimizing ML Pipelines. An ML pipeline is a collection of computational *tasks* that produce and consume *artifacts*. Sharing computations, a.k.a. common subexpression elimination, is a widespread practice for optimization that merges multiple executions into one to remove redundant tasks. It identifies exact sequences of tasks applied to the same data. Reuse is a technique that builds on the idea of common subexpression elimination. It identifies artifacts computed in the past and, if stored, decides if it is beneficial to load or recompute them.

The capacity to reuse highly depends on which previously computed artifacts are stored. As a result, *reuse* is always accompanied by a *materialization* strategy, which focuses on which of the produced artifacts should be stored. State-of-the-art research on the reuse-materialization problem in ML pipelines [4], [5], [6], [7] showcases significant benefits.

Task and Artifact Equivalence. Equivalent subexpressions are sequences of tasks that, although not identical, produce the same results. They have been studied in the fields of relational databases, batch or streaming data platforms [8], [9], or analytical workflow engines [10]. Equivalences exist: (a) *Among tasks*, when different physical implementations of a logical operator exist, e.g., a join can be implemented as merge-sort, nested loops, or hash-join; when the same algorithm can be implemented in different programming languages, libraries, or frameworks, e.g., implementation of k-means in scikit-learn, PyTorch, TensorFlow, cuML. (2) *Among artifacts*, when different sequences of tasks produce the same result [11], [10].

Our Contributions. While existing ML pipeline optimization techniques are based on the concepts of reuse and materialization, they fail to consider the impact of equivalences. Our work addresses this gap, making a series of contributions.

Novel Pipeline Representation. We acknowledge ML tasks are multi-input and multi-output and have complex dependencies as one task requires the output of others. Directed *hypergraphs* are a natural fit as a representation as they provide a one-to-one mapping of multi-input and multi-output ML tasks to *hyperedges* and artifacts to nodes. Moreover, we explicitly represent in the hypergraph the *state* of ML operators that can be fitted, like preprocessing steps and ML models, which, as we present

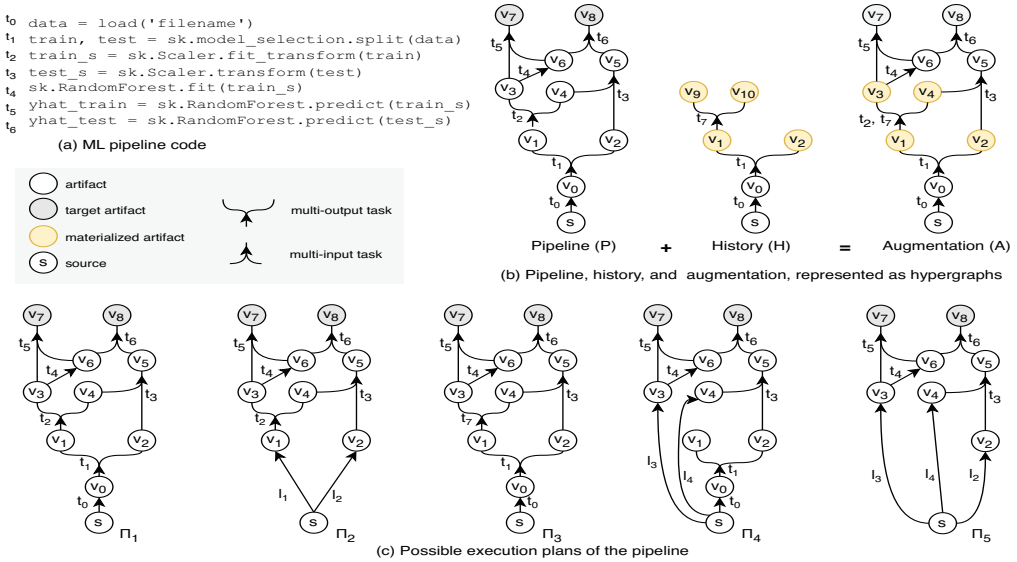


Fig. 1: Overview of the pipeline optimization process in HYPPO. (a) A user seeks to execute an ML pipeline, described here in Python-like language. (b) HYPPO represents the pipeline as a hypergraph P. Based on a history H capturing past pipeline executions, HYPPO generates an augmented pipeline A. (c) This augmentation exploits equivalences to encode alternative execution plans for the original pipeline. HYPPO explores the optimization space to identify the plan of minimum cost. While *materialization* enables opportunities for artifact reuse, resulting here in plans Π_1 and Π_2, *equivalences*, here between current task t_2 and past t_7, reveals additional optimization opportunities as captured in plans Π_3–Π_5.

in the evaluation, are an excellent candidate for materialization. In addition, hypergraphs are the appropriate abstraction to capture alternative, equivalent ways to produce artifacts. Equivalence introduces OR semantics: *any* alternative way to produce an artifact suffices. In a hypergraph, a hyperedge/task has multiple input nodes/artifacts annotating its dependencies, and alternative computations can be annotated by multiple incoming hyperedges/tasks to an artifact. Note that directed graphs (DAGs), used in prior work [4], [5], cannot represent equivalences. DAGs annotate computation dependencies as multiple incoming edges (e.g., a join operator depends on more than one artifact), making them unable to represent alternative computations where an artifact may not be tied to one set of dependent artifacts. In other words, DAGs encode AND semantics: producing an artifact with multiple incoming edges requires producing *all* its dependent artifacts. Capturing both AND and OR semantics with DAGs is not possible, as we cannot differentiate between edges that represent alternatives and edges that together capture dependencies.

Example 1. *Figure 1(a) shows an ML pipeline based on scikit-learn (sk) and coded in Python-like syntax, implementing a series of tasks: (t_0) load a dataset, (t_1) split it into train and test datasets, (t_2) use the train data to fit a standard scaler, (t_3) apply the scaler to the test data, (t_4) fit a random forest classifier to the train data, and finally, compute predictions for the train (t_5) and the test (t_6).*

This pipeline can be encoded as the hypergraph depicted in Figure 1(b-left). We employ a special node s*, called source, to represent storage locations; all load tasks start from* s*. Observe, for example, that t_1 is a multi-output task represented as a hyperedge connecting one (data v_0) to two artifacts (train and test v_1, v_2). Interestingly, fit-like tasks (t_2, t_4) have a state that is explicitly captured as an artifact (v_4, v_6). Therefore, tasks, such as transform and predict, that utilize this state are represented as multi-input hyperedges, e.g., t_3, t_5, t_6.*

Logical Equivalences. We acknowledge that ML pipelines often include different implementations of the same logical operators. This leads to uncovering additional opportunities for reusing artifacts from past iterations or experiments beyond what standard reuse-materialization can offer.

Example 2. *The idea of reuse-materialization is to log historical executions and keep a record of materialized artifacts. Figure 1(b-middle) depicts the history using our hypergraph notation and highlights the materialized artifacts. In the past, a task identical to t_1 was executed, and its output, which is identical to v_1, v_2, was decided to be materialized; for convenience, we use the ids of the current pipeline (i.e., t_1, v_1, v_2) to represent this fact in the history.*

Assume now that the past task t_7 performed standard scaling of the train data v_1, just like t_2, but using a function in another framework (say TensorFlow). Since t_2 and t_7 are different implementations of the same logical operator, they are equivalent tasks*, and thus produce* equivalent artifacts *($v_9 \equiv v_3$, $v_{10} \equiv v_4$) when fed with the same input (v_1).*

Generation of an Execution Plan. In contrast to prior work, we consider possible execution plans that exploit both reuse opportunities and task/artifact equivalences. We identify part of the history that is relevant in terms of reuse and equivalence

with the current pipeline. Thus, we enrich the pipeline with the history to create an augmented one. The augmentation is also represented as a hypergraph, with the important difference that some artifacts might have multiple incoming hyperedges, representing alternative ways to derive them: (a) from storage if materialized, and (b) from equivalent tasks (or sequences thereof). The state of the art on reuse-materialization [4], [5] partially exploits the former optimization opportunities since they reuse identical but not equivalent artifacts.

Example 3. *Figure 1(b-right) depicts the augmented pipeline. Here, we only highlight materialized artifacts instead of depicting the load tasks to avoid cluttering the hypergraph; e.g., artifact* v_1 *has actually two incoming hyperedges, the depicted task* t_1 *and the non-depicted load task* l_1 *connecting* s *to* v_1.

The discovered equivalences between current task t_2 *and artifacts* v_3, v_4, *and previous task* t_7 *and artifacts* v_9, v_{10}, *are represented by the two parallel hyperedges* t_2 *and* t_7 *in the augmentation. This encodes that to derive* v_3 *or* v_4, *there are three possibilities: execute the current task* t_2, *re-execute the past task* t_7, *or load from storage.*

The augmentation encodes all possible ways to produce the required artifacts or *targets* of the pipeline at hand. We define the notion of an execution *plan*. Intuitively, a plan is a hypergraph such that (a) it contains the target nodes, and (b) every artifact it contains can be derived (computed or loaded) by executing a set of tasks starting from the source. Assuming a model that provides estimates (e.g., by utilizing observations of past executions) of the cost (in terms of time or money) of executing each task, we formulate and study the search problem of finding the plan of minimum cost.

While the problem of reuse has been previously formulated as a MAX-FLOW problem [4], the addition of alternative computations transforms the problem into a computationally harder search problem in directed hypergraphs. In fact, the problem is NP-hard; finding a plan of minimum cost connecting the source to a set of targets in a directed graph is the directed variant of the Steiner tree problem. In practice, however, the size of ML pipelines is moderate. A recent study on large collections of pipelines residing on GitHub reports that most pipelines had an average length of four, while a few extreme pipelines contained up to a few dozens of tasks. Hence, the hypergraphs typically derived from such ML pipelines would also be moderately small, indicating that devising practical algorithms to find optimal plans is feasible.

Example 4. *Figure 1(c) presents a set of plans* Π_1–Π_5 *extracted from the augmentation.* Π_1 *is the original pipeline, while* Π_2 *chooses to load materialized artifacts* v_1 *and* v_2. *Note that previous work based on reuse-materialization can only choose among* Π_1 *and* Π_2. *Plans* Π_3–Π_5 *are an option thanks to the revealed task and artifact equivalences. For example,* Π_3 *chooses to execute task* t_7 *instead of* t_2, *while* Π_5 *loads materialized artifacts* v_2–v_4. *Assuming the combined cost of loading tasks* l_2–l_4 *is lower than that of other options to derive artifacts* v_2–v_4, *plan* Π_5 *is the optimal.*

The aforementioned contributions are manifested in the design of our HYPPO system, short for Hypergraph Pipeline Optimizer. The remainder of this paper is structured as follows. Section II overviews related work. Section III introduces the concepts and formulates research problems. Section IV presents HYPPO. Section V presents a detailed experimental evaluation of HYPPO. Section VI concludes the paper.

II. Related Work

There exists a sheer amount of systems to manage the ML lifecycle, including industry-scale implementations such as Kubeflow [12], Apache SystemDS [13], AzureML [14] or MLflow [15]. These systems primarily focus on tracking pipeline executions and storing metadata, but they do not address optimization aspects of exploratory analysis, such as the reuse of artifacts among different pipelines. There are two surveys on ML lifecycle management systems [16], [17].

Recent trends in AutoML focus on finding performant pipelines [18], [19], [20] or models [21], [22] by reducing the number of pipelines or models proposed for evaluation [3]. These approaches are orthogonal to our method since they select a set of pipelines or models for evaluation, while our work optimizes the execution time across multiple runs.

a) Materialization and reuse in EML: Lima [6] offers detailed lineage tracking and reuse within a single pipeline. However, it does not reuse artifacts between different pipelines.Studies on feature selection [7] and model databases [23] have also explored Materialization-Reuse. MISTIQUE [24], Helix [4], and Collab [5] focus on artifact materialization and reuse across pipelines and are closer to our work. Helix and Collab present reuse and materialization techniques within set storage limits to minimize costs across pipelines. Helix tackles the optimal reuse plan as a solvable project selection problem [25] using polynomial-time algorithms. Collab employs a linear algorithm, however, at the expense of not always yielding the best solution. In terms of materialization policies, Helix is restricted to artifacts from the immediate preceding pipeline, while Collab expands this by creating an experimental graph that encompasses materialized artifacts from all prior pipelines.

b) Operator Equivalence: The concept of multiple physical implementations for equivalent logical operators has been a staple in relational query optimization since its early days, as seen in the Volcano optimizer [26] and in query rewriting techniques [27]. This principle has since been applied across various domains, including Video Analysis [28], optimization of User-Defined Functions (UDFs) [29], [30], [31], [32], analytical workflows [33], and streaming big data applications [9]. Raven [10] has been developed for identifying equivalent workflows, although it lacks a decision-making cost model. In the realm of ML pipelines, KeystoneML [34] interprets a logical sequence of operators and assigns a physical operator to each. This process involves a cost model for every physical operator and entails evaluating every potential physical implementation on a subset of the data before running a pipeline.

III. Problem Formulation

A. ML Pipelines

An ML pipeline comprises a series of *logical operators*. For example, a typical classification pipeline includes a StandardScaler operator that standardizes features by subtracting the mean and scaling to unit variance. A logical operator can be implemented in different ways, e.g., programming languages (Python, Scala, Java, R, C#, etc.), ML frameworks/libraries (scikit-learn, TensorFlow, PyTorch, JAX, Spark MLlib, H20, RAPIDS, ML.NET etc.), algorithms (GPU/CPU optimizations). We refer to these implementations as *physical operators*. Some ML operators are configurable and accept hyperparameters such as number of clusters, learning rate; we refer to the set of hyperparameters and values collectively as the *configuration* of an operator.

Physical ML operators expose API methods, which we call *tasks*, and have an internal state, which we call *op-state*. There exist some fundamental tasks that are common across physical implementations; we call these *task types*. For example, preprocessing operators like StandardScaler expose *fit* and *transform* tasks. A *fit* task computes the StandardScaler's state, which comprises the mean and standard deviation of the training data. A *transform* task employs these calculated values to standardize features in both the training and test data. ML models, such as linear regression, decision trees, and neural networks, have a *fit* task that determines their parameters/weights, i.e., their op-state, and a *predict* task that uses these parameters/weights to make inferences/predictions.

Thus, a task consumes and produces one or more *artifacts*, which can be: (a) *data*, including datasets with a schema analogous to DataFrame objects [35] or NumPy Arrays [36], and values or collections thereof; or (b) an *op-state* of some operator, such as a pre-trained model.

B. A Hypergraph Representation

Past approaches represent ML pipelines as directed acyclic graphs, either considering single-input or multiple-input operators (e.g., join). However, they fail to represent a large variety of ML operations that are multiple-output as well. In this work, we model them as directed hypergraphs (DH) [37], a natural generalization of digraphs that have many applications, including representing functional dependency in databases [38].

A *directed hypergraph* $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ consists of a set of nodes V and a set of hyperedges $\mathsf{E} \subseteq 2^{\mathsf{V}} \times 2^{\mathsf{V}}$; we also write $\mathsf{V}(\mathsf{G})$ and $\mathsf{E}(\mathsf{G})$ to refer to the nodes and hyperedges of G. A *hyperedge* $\mathsf{e} = (\mathsf{tail}(\mathsf{e}), \mathsf{head}(\mathsf{e})) \in \mathsf{E}$ connects a set of nodes $\mathsf{T}(\mathsf{e})$, called the *tail*, to a set of nodes $\mathsf{head}(\mathsf{e})$, called the *head*, where $\mathsf{tail}(\mathsf{e}), \mathsf{head}(\mathsf{e}) \subseteq \mathsf{V}$. A directed hypergraph reduces to a directed graph when the head and tail of each hyperedge consist of a single node.

A subhypergraph of some DH, is defined naturally. Given a DH $\mathsf{G} = (\mathsf{V}, \mathsf{E})$, we say that $\mathsf{G}' = (\mathsf{V}', \mathsf{E}')$ is a *subhypergraph* of G if $\mathsf{V}' \subseteq \mathsf{V}$, $\mathsf{E}' \subseteq \mathsf{E}$, and $\mathsf{E}' \subseteq 2^{\mathsf{V}'} \times 2^{\mathsf{V}'}$.

In a directed hypergraph, a node v can be in the head and tail of multiple hyperedges, similar to how a node in a directed graph can have multiple incoming and outgoing edges. The *backward* (resp. *forward*) *star* of v is the set of edges that have v in its head (resp. tail), i.e., $\mathsf{bstar}(\mathsf{v}) = \{\mathsf{e} \in \mathsf{E} : \mathsf{v} \in \mathsf{head}(\mathsf{e})\}$ and $\mathsf{fstar}(\mathsf{v}) = \{\mathsf{e} \in \mathsf{E} : \mathsf{v} \in \mathsf{tail}(\mathsf{e})\}$.

An important concept in graphs is connectivity. There are several concepts that generalize the notion of connectivity/reachability to directed hypergraphs. The concept relevant to ML pipelines is B-connection, and is defined recursively [39]. Given a DH G, we say that a node t is *B-connected* to a node s if (a) $\mathsf{t} \equiv \mathsf{s}$, or (b) there exists a hyperedge $\mathsf{e} \in \mathsf{E}(\mathsf{G})$ that has t at its head $\mathsf{head}(\mathsf{e})$ and all nodes in its tail $\mathsf{tail}(\mathsf{e})$ are B-connected to s. We extend this definition to a set of nodes S. Given a DH G, we say that S is *B-connected* to a node t if a dummy node s is B-connected to t in the extended graph G', having $\mathsf{V}(\mathsf{G}') = \mathsf{V}(\mathsf{G}) \cup \{\mathsf{s}\}$ and $\mathsf{E}(\mathsf{G}') = \mathsf{E}(\mathsf{G}) \cup \{(\mathsf{s}, \mathsf{S})\}$.

C. Definitions

In the following, we present the main concepts related to ML pipelines, and we define them using hypergraphs.

1) Pipeline: An ML pipeline is represented as a labelled hypergraph, called the *pipeline* and denoted as P.

Definition (Pipeline)**.** *A pipeline* P *is a labelled DH, where nodes are artifacts and the hyperedges represent the tasks producing and consuming the artifacts. A node is labelled with name, type, and size. A hyperedge is labelled with name, type, and configuration.*

Note that each hyperedge in P represents a task and can hence have multiple inputs and outputs. As a convention, we use a special node called the *source* s to represent all possible storage locations. Any artifact v that can be loaded by the pipeline is connected to s with a hyperedge of type 'load'.

2) Equivalence: Physical operators corresponding to the same logical operator are considered *equivalent*. For example, torch.pca_lowrank and sklearn.decomposition.PCA are equivalent implementations of the same logical operator, the PCA decomposition. Following past work, we consider two tasks *equivalent* when: (a) they have the same input; and either (b) they correspond to the same logical operator and are of the same task type, or (c) given the same input, they produce identical results [40], [41], [42], [43], [44]. Moreover, two artifacts are *equivalent* when they are produced by equivalent tasks. Two notes are of interest here. First, the task equivalence definition assumes that tasks are deterministic. However, in certain use cases, it might make sense to consider the output of stochastic tasks to be equivalent; e.g., the training of a model under the same hyperparameters. Second, while ML pipelines typically use consistent data formats for input and output (e.g., DataFrame, NumPy arrays), cross-framework libraries may use distinct intermediate representations (e.g., Apache Beam's PCollection). In this case, we acknowledge that the equivalence of artifacts comes with some translation overhead.

3) Cost of Tasks: We consider two types of costs: computational time (e.g., milliseconds) and monetary (e.g., Euros). The cost $\mathsf{time}(\mathsf{e})$ of task e represents its execution time if it is a computational task or the retrieval time if it is a task that

loads an artifact from some storage (represented as the source node s). The monetary cost $\mathsf{price}(e)$ of task e relates to typical cloud charging schemes and involves computing and storage:

$$\mathsf{price}(e) = \mathsf{time}(e) \times \mathsf{price\_per\_time\_unit} + \sum_{v \in \mathsf{tail}(e)} \mathsf{size}(v) \times \mathsf{price\_per\_size\_unit}$$

4) History: The history, denoted as H, archives the knowledge acquired by previous pipeline executions. That is, it keeps track of all artifacts and tasks, along with their metadata and execution statistics, used by running pipelines.

Definition (History)**.** *The history H is a labelled directed hypergraph. Nodes and hyperedges model artifacts and tasks, respectively. A labelled node contains metadata as artifact name, cost, type, size, access frequency, and version. A labelled edge comprises metadata as task name, cost, and type.*

The history records all observed artifacts and tasks that created those artifacts, along with their lineage and useful information such as their cost and whether they are materialized. We use the annotation $e.\mathsf{cost}$ to refer to any kind of cost related to task e. Note that in H, the backward star of a node indicates alternative ways to obtain the artifact, and its forward star represents all tasks that depend on this artifact.

A new pipeline can be compared against H, exploring reuse execution opportunities (by traversing the hypergraph) and leveraging materialized artifacts (by following the pointers of the hypergraph to storage). In this sense, the history serves as a *dual cache* for ML pipeline execution.

5) Plan: A plan is a special type of directed hypergraph that corresponds to an *executable* ML pipeline.

Any ML pipeline describes the partially ordered series of tasks that need to be executed to produce the required artifacts, which we call *target nodes* T. In the hypergraph representation, these are sink nodes that have empty forward stars, i.e., there is no downstream task that uses them.

Definition (Plan)**.** *Given hypergraph G, a set of sources S and a set of targets $T \subseteq V(G)$, an S-T plan Π_{ST} is a minimal (with respect to deletion of nodes and hyperedges) subhypergraph of G such that each target $t \in T$ is B-connected to S.*

Note that we are typically interested in plans that have as source nodes the node s representing the storage locations. A plan has the following properties. (a) For every hyperedge e it includes, all nodes in its head are reachable (i.e., B-connected) from the source nodes. This implies that a plan can be executed as for each task, there is always a way to generate its required artifacts. (b) A plan contains all target nodes but does not contain all source nodes. In an ML pipeline, the intuition is that we want to compute all target artifacts T given the available artifacts S, while some source nodes may not be utilized.

D. Problem Statement

The problem of optimizing ML pipelines consists of two interrelated sub-problems, corresponding to the *reuse* and *materialization* tasks, respectively. In the first sub-problem, called Plan, given an ML pipeline P and the history H comprising the accumulated knowledge of past pipeline executions, the task at hand is to identify the optimal plan Π to execute the input pipeline by leveraging recorded information in the historical hypergraph. To solve this, we employ a novel optimization algorithm operating on our novel hypergraph representation, that enables us to replace tasks, artifacts, or subhypergraphs, of the input pipeline with equivalent, more beneficial tasks, artifacts, or subhypergraphs or bypassing them altogether by reusing a previously materialized set of artifacts. In the second sub-problem, called Materialize, given a newly executed plan Π, the history H, and a storage budget B, the task is to identify which artifacts from Π to materialize and potentially which materialized artifacts stored H to evict so that (a) the total size of stored artifacts does not exceed B, and (b) the cost of future pipeline executions is minimized.

1) Identify Plan: Given a labelled hypergraph G (such as a plan), we define its cost, denoted by $\mathsf{cost}(G)$, as the sum of the costs of its edges: $\mathsf{cost}(G) = \sum_{e \in E(G)} e.\mathsf{cost}$. We formulate the planning problem for a pipeline requesting targets T:

Problem 1 (Plan)**.** *Given a weighted DH H, the source s, and a set of terminal nodes T, find a plan Π_{sT} from s to T of minimum cost.*

It is interesting to relate this problem to existing graph search problems. If the hypergraph is a directed graph, and $T = \{t\}$ is a single target, a minimum cost plan is simply the shortest path from s to t. If the hypergraph is a directed graph, a minimum cost plan is a directed Steiner tree connecting the root s to terminal nodes T [45]; finding one is NP-hard, and the decision variant for undirected graphs was one of the original 21 NP-complete problems. If the target is a singleton, i.e., $T = \{t\}$, a minimum cost plan corresponds to a minimum cost B-hyperpath from s to t [39]; finding one is NP-hard [39].

2) Identify Materialization Strategy: We aim at identifying the artifacts to materialize in the presence of a constraint on the storage budget B, given that we can compute how often the artifacts will be accessed, that we can estimate their size, and that we can estimate the runtime associated with materializing them (this information is maintained in the history H). Hence, the materialization problem is formulated as follows: given a storage budget B, we want to find the set of artifacts to materialize that minimizes the cost for future pipeline executions. This is an NP-hard combinatorial optimization problem. Related work considers heuristic algorithms employing a benefit function for each artifact as the reduction in execution time it could potentially lead to [5], [4]. In our current implementation, we follow a different approach.

Let $v \in V(H)$ be the artifact we examine to materialize or not. The decision depends on the *savings benefit* $\mathsf{gain}(v)$ of the said artifact. We employ a *goodness measure* to choose what materialized artifacts we should prefer [46]. Various criteria may be used to define goodness, such as (a) the time an artifact was last accessed, which resembles a Least Recently Used (LRU) eviction policy, (b) the frequency of access for an artifact, i.e., Least Frequently Used (LFU) policy, (c) the

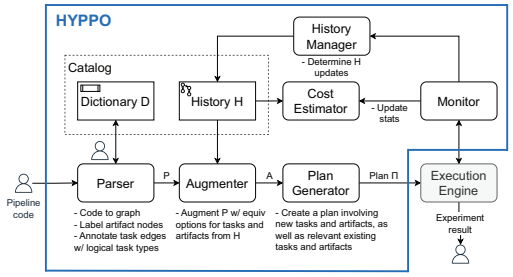


Fig. 2: HYPPO's system architecture

size of an artifact in disk size, keeping larger artifacts in place, i.e., Smaller-Fragment-First (SFF) eviction policy. Currently, we consider a replacement strategy that shows excellent results in our experiments and involves a combination of factors, including: (a) artifact's loading cost from storage, $\mathsf{load}(v)$, (b) cost of re-computing an artifact for a future pipeline (minimum cost of a plan s-t), $\mathsf{cost}(v)$, and (c) frequency of access, $\mathsf{freq}(v)$. In particular, for an artifact v, $\mathsf{gain}(v)$ is defined as the expected penalty of re-producing the artifact, if it is evicted, amplified by its popularity (access frequency), and normalized by its loading time, which resembles a Smaller Penalty First (SPF) replacement policy: $\mathsf{gain}(v) = \frac{\mathsf{freq}(v) \times \mathsf{cost}(v)}{\mathsf{load}(v)}$. We use as an estimate of $\mathsf{cost}(v)$ the cost of re-computing the artifact v.

In addition, a related study reveals that most changes in a pipeline occur after the pre-processing stage [47]. Hence, there is an increased probability of pipeline reuse for artifacts that are closer to the source. This decays at a rate proportional to the *depth* of the artifact v, which is calculated as the average number of hyperedges away from the source, to account for the alternate ways (e.g., different plans, with and without materialization) to obtain v. Hence, we can prioritize such artifacts for materialization with a plan locality coefficient: $\mathsf{pl}(v) = 1/e^{\frac{1}{\mathsf{depth}(v)}}$, where $v \in H(V)$ is an artifact and s is the root of H. Hence, gain is weighted with pl as: $\mathsf{pl}(v) \times \mathsf{gain}(v)$.

We formulate the problem of identifying the most beneficial materialization strategy as follows.

Problem 2 (Materialize)**.** *Optimize:*

$$\max_{W} \sum_{v \in V(H)} 1/e^{\frac{1}{\mathsf{depth}(v)}} \times \frac{\mathsf{freq}(v) \times \mathsf{cost}(v)}{\mathsf{load}(v)}$$

$$\textit{subject to} \quad \sum_{v \in V(H)} (\mathsf{size}(v) \times w_v) \leqslant B$$

where B *is a given storage budget and* $w_v \in \{0, 1\}$ *is a indicator variable signifying if an artifact is materialized* (1) *or not* (0).

IV. The HYPPO System

A. Architecture

The core components of HYPPO's architecture, illustrated in Figure 2, are: (a) catalog, (b) parser, (c) equivalence verifier, (d) plan generator, (e) cost estimator, (f) monitor, and (g) graph manager. The pipeline optimization process starts with the user submitting their ML code. The parser consults the catalog's dictionary to create a *pipeline* representation P from the raw code. The augmenter then retrieves relevant historical information from the *history* H to derive the *augmentation* A, which encodes alternate options to generate artifacts in the original pipeline. The plan generator retrieves estimations for the cost of tasks and searches the space of alternatives included in the augmentation to derive the optimal execution *plan* Π. The plan is then executed, and the monitor collects important metrics to feed the cost estimator. Finally, the graph manager decides which artifacts to materialize and updates their history. Next, we discuss the components of HYPPO.

B. Catalog

The catalog comprises the task dictionary D and history H.

a) Dictionary: The task *dictionary* contains an extensible set of equivalent operators and tasks typically met in ML pipelines. The dictionary entries follow the form: $D = \{\ldots, \mathsf{lop}_i.\mathsf{tasktype}_j : [\ldots, \mathsf{impl}_k, \ldots], \ldots\}$, where a task type of a logical operator lop is associated with equivalent physical implementations impl; the dictionary entries also capture operator configurations—e.g., $\mathsf{Ridge}(\mathsf{alpha} = 75.0)$.

Every logical operator must be associated with at least one physical operator [48]. Logical operators with multiple physical implementations are candidates for optimization. For example, the 'fit' task of the logical operator PCA has equivalent physical implementations in scikit-learn and in PyTorch. Hence, the dictionary would have an entry $\mathsf{PCA.fit} : [\mathsf{sklearn.decomposition.PCA.fit}(), \mathsf{torch.pca\_lowrank}()]$. In our current implementation, the dictionary contains 40 operators such as various ensemble, evaluation, regression, imputation, scaler, PCA, and SVM methods. The dictionary can be straightforwardly extended to support additional operators.

b) History: The history is the labelled hypergraph H described in Section III-C4. The history contains the special node s that represents artifact sources. The history thus also contains hyperedges of type 'load' that connect s to each artifact that can be retrieved either locally (i.e., is a materialized artifact) or remotely from some storage location. As history continuously grows, for practical considerations, not all its artifacts remain materialized. The amount of artifacts stored is controlled by a storage *budget* B and is maintained within the available quota with appropriate eviction strategies. An example of history is shown in the middle panel of Figure 1(b), where four artifacts are materialized.

C. Parser

The parser processes the pipeline code and converts it to the *pipeline* labelled hypergraph P. In doing so, it assigns names to artifacts that encode equivalences that can be later exploited.

First, for each function call in the pipeline code, the parser identifies the logical operator and task type it corresponds to by probing the dictionary. If a task cannot be mapped to a known operator in D, HYPPO treats it as an operator with a single physical implementation (the one provided in the pipeline code). After doing so, the parser creates the hypergraph P that corresponds to the input pipeline. The parser annotates each task with the name of the logical operator and task type.

An important step of the parser is the annotation of artifacts. Each artifact v gets a name that encodes its backward star in the pipeline using logical operators and task types. For example, if this is a sequence of hyperedges/tasks e_1, e_2, the artifact name would be e_1.lop_e_1.tasktype_e_2.lop_e_2.tasktype. In practice, names are converted to hashes of fixed size. The parser also determines the required artifacts, the *targets*, which are "sink" nodes with empty forward stars. As an example, the code in Figure 1(a) is parsed to produce the pipeline presented in the left panel of Figure 1(b).

D. Pipeline Augmenter

The function of the augmenter is to enrich the pipeline P with alternative, equivalent options for task and artifact computation that have been recorded in the history H. This enables a larger space of potential computation and execution options, allowing the HYPPO optimizer to pick a presumably beneficial plan. The outcome of this process is an augmented pipeline, or simply an augmentation A that is a DH with the property that the pipeline P is a subhypergraph of A. Compared to the pipeline, the augmentation contains the part of the history H that B-connects s to all artifacts in H that are equivalent to those in P. Equivalent artifacts are immediately apparent thanks to our naming convention, which captures the recursive definition of artifact equivalence.

The augmentation copies the set of targets (required artifacts) from the pipeline. The edges in the augmentation that are not in the history, i.e., those in $E(A) \backslash E(H)$, are called *new tasks*, as they have not been recorded so far. When HYPPO operates in the *exploration mode* (discussed shortly), then the set of targets also contains those artifacts that are at the head of the new tasks. The right panel of Figure 1(b) shows the result of the augmenter applied to the pipeline and history depicted in the other two panels.

E. Plan Generator

The plan generator solves Problem 1. It uses the cost estimator to derive the cost of every task in the augmentation A. Thus, each hyperedge $e \in E(A)$ is labelled with a positive weight $e.cost$ capturing the estimated cost of executing task e.

The plan generator seeks to identify an "optimal" plan among those encoded in the augmented pipeline A. Specifically, the plan should optimize for the total execution cost (either computation or monetary). This is done by reusing materialized artifacts, as well as exploiting equivalent alternative tasks to derive the artifacts contained in the original pipeline. A plan may avoid executing some of the new tasks (under the exploitation mode, discussed shortly), when they can be exchanged by cheaper equivalent ones, or sequences thereof.

While Problem 1 is NP-hard in its general case, the size of the augmented pipeline A is typically small. Thus, it is realistic to search for the optimal plan; we show that the running time of our algorithms is less than ten milliseconds. In what follows, we present an exact algorithm for the problem at hand and several variations. Lastly, we present the complexity analysis and address any scalability issues.

Algorithm 1 presents the main algorithm OPTIMIZE, which depends on the EXPAND procedure presented in Algorithm 2. OPTIMIZE searches for a plan starting from the targets T and proceeds traversing hyperedges backwards (from head to tail) until it reaches the source s. Doing so, it maintains a list of incomplete plans; an incomplete plan Π is a plan from some set of source nodes $\Pi.frontier$ to the target T. As the algorithm proceeds, each incomplete plan is *expanded* until it becomes a complete s-T plan or is pruned.

The algorithm groups incomplete plans in a data structure Q, initially populated with the trivial plan from T to T containing no hyperedges (lines 2–3). Then, the algorithm performs a series of operations until Q is empty (lines 4–11). At each iteration, an incomplete plan Π is selected and removed from Q (line 5). If it has a cost greater than the best seen for a complete plan, it is pruned (line 6). If the source of Π contains no nodes other than s, Π is a complete plan (line 7), and, since it was not pruned, it is the best plan seen; thus, the algorithm records it and its cost (line 8). Otherwise, Π is incomplete and expanded, leading to a set of plans $\boldsymbol{\Pi}$ inserted into Q.

We next discuss the EXPAND procedure. The goal is to expand a given plan Π and generate new plans. The *frontier* of a Π contains its sources and some additional nodes to explore. Each expansion of Π is defined by a *move*, which is a set of hyperedges to follow from nodes in $\Pi.frontier$ (lines 3–5). More precisely, a move is a set of hyperedges such that it contains exactly one hyperedge from the backward star of each frontier node. A single-move expansion (lines 6–14) of plan Π is the plan Π' that results from p by performing the move. During the construction of Π', each hyperedge of the move is examined (lines 8–14). Specifically, the hyperedge and its incident nodes are added to Π', and its cost is updated (lines 9–11). Then, the algorithm updates (lines 12–14) the node sets $visited$ and $frontier$, whose function is to avoid cycles, and determine the set of nodes to expand next, respectively.

Complexity Analysis and Scalability. The complexity of our algorithm can be expressed as a factor of (a) the number ℓ of nodes in the longest path of the augmented graph; (b) the maximum number f of nodes in the frontier; and (c) the maximum number m of alternatives for an artifact, i.e., the maximum number of incoming hyperedges. For each plan Π examined in the main loop of OPTIMIZE, the EXPAND procedure generates $O(m^f)$ new plans (each of the f frontier nodes has m alternatives). Because the longest path is ℓ, OPTIMIZE explores a search tree of depth ℓ and of fanout $O(m^f)$, resulting in a worst-case complexity of $O(m^{f \cdot \ell})$.

An exhaustive search requires navigating over the combination of all possible hyperedges (one per node). If we denote as n the number of nodes in the graph, then the complexity of exhaustively searching for all of the valid plans in the augmented graph is $O(m^n)$. Yet, recall that not all combinations lead to minimal B-connected paths (executable plans), an aspect that our algorithm exploits by examining only minimal B-connected paths. This is possible using an inverse traversal of the graph pruning any node that does not

Algorithm 1: OPTIMIZE

```
   Input: augmentation A; source s; target nodes T
   Output: plan of minimum cost Π* = (V_Π*, E_Π*)
1  cost* ← ∞; Π* ← none
2  Π_0.cost ← 0; Π_0.visited ← ∅; Π_0.frontier ← T; Π_0.plan ← ∅
3  Q.insert(Π_0)
4  while Q ≠ ∅ do
5      Π ← Q.select()                                  ▷ pick a plan from Q
6      if Π.c ⩾ c* then continue                       ▷ ignore this plan
7      if Π.frontier = {s} then                        ▷ if plan is an s-T plan
8          cost* ← Π.cost; Π* ← Π
9      else
10         Π ← EXPAND(A, Π, s)                         ▷ expand this plan
11         foreach Π ∈ Π do Q.insert(Π)                ▷ add all expansions to Q
12 return Π*
```

Algorithm 2: EXPAND

```
   Input: augmentation A; plan Π; source s
   Output: set Π of plans expanded from Π
1  Π ← ∅
2  foreach v ∈ Π.frontier\s do
3      E_v ← bstar(v)                  ▷ set of options (hyperedges) to explore
4  M ← ×_{v∈Π.frontier\s} E_v          ▷ cross product of sets of options
5  foreach move ∈ M do                 ▷ create a new plan adding move to Π
6      Π' ← copy(Π); Π'.frontier ← ∅
7      foreach e ∈ move do
8          newNodes = head(e)\Π'.visited
9          if newNodes ≠ ∅ then
10             Π'.cost ← Π'.cost + e.cost
11             Π'.visited ← Π'.visited ∪ (newNodes)
12             Π'.frontier ← Π'.frontier ∪ (tail(e)\Π.visited
13             Π'.plan ← Π'.plan ∪ {e}
14     Π ← Π ∪ {Π'}
15 return Π
```

participate in the production of T. Such pruning of the search space allows for a more efficient exploration –with a worst-case complexity of $O(m^{f \cdot \ell})$– than that of an exhaustive search.

Practical considerations. While n can increase rapidly, it has been reported that, in practice, ℓ is usually bounded. A recent study reveals that in practice the typical length of pipelines is between 4–15, with only a handful of pipelines ($\sim$10 among $\sim$10K) reaching a length of about 50 [49]. Hence, we can devise practical optimization techniques that would improve the overall performance of our approach.

The influence of factor ℓ can be reduced by using guided search. This is achieved by changing the data structure Q, which defines the order in which plans are examined. We implement Q as either a LIFO stack, denoted as OPTIMIZE-STACK, or a priority queue with a key to the incomplete plan's cost, denoted as OPTIMIZE-PRIORITY. While both OPTIMIZE-PRIORITY and OPTIMIZE-STACK have the same worst-case complexity as exhaustive search, in practice, as we show in our scalability study, this does not happen. A direction for future research is to define an A*-like heuristic that estimates the minimum cost of expanding an incomplete plan to the source. The influence of factor m can be reduced by populating the augmented graph with only potentially beneficial alternatives, while that of f can be reduced by creating individual plans for each request and combining them. However, this approach could lead to suboptimal plans.

Accuracy can be sacrificed for a potential "good" plan to be discovered in time linear to the number of hyperedges. Using a greedy approach, the expand function can be changed to return a single *move*, the *move* with the minimum cost. Given the frontier, follow the minimum cost hyperedge of each artifact in the frontier. This approach can greatly reduce the complexity since each node and hyperedge in the graph is visited at most once, resulting in a worst-case complexity of $O(n + m \cdot n)$.

Exploration vs. Exploitation. The plan generator may operate in either *exploration* or *exploitation* mode. In the former, HYPPO eagerly tries to learn more information, and thus, it promotes the execution of new tasks, even if there is a clearly cheaper plan to follow. On the other hand, in exploitation mode, it leverages prior knowledge for plan generation.

In exploitation mode, OPTIMIZE is executed as is. In exploration mode, we want the plan to include the new tasks. To force this, we make some changes to the initialization step (line 2). For each new task e, we (a) add this edge to the plan, i.e., $V_{\Pi_0} \leftarrow V_{\Pi_0} \cup \text{head}(e) \cup \text{tail}(e)$, $E_{\Pi_0} \leftarrow E_{\Pi_0} \cup \{e\}$, and (b) update visited and frontier nodes as $\Pi_0.\text{visited} \leftarrow \Pi_0.\text{visited} \cup \text{head}(e)$, $\Pi_0.\text{frontier} \leftarrow \Pi_0.\text{frontier} \cup \text{tail}(e)$.

As both modes serve a practical purpose, in our implementation, we steer the plan generator in either direction with a tunable knob mo, defined as $\text{mo} = \lfloor \#\text{new\_tasks} \times c_{exp} \rceil$ with c_{exp} taking values in the range $[0, 1]$ where 0 and 1 indicate a preference towards exploitation and exploration, respectively.

F. Monitor

HYPPO's monitor serves two functions. First, it monitors pipeline execution and collects traces of metrics such as resource utilization and execution time. These are used by the cost estimator to update the statistics maintained for the operators listed in the dictionary. Second, it monitors the execution of new tasks and the cost of producing the resulted artifacts. This information is used by the graph manager.

G. Cost Estimator

The cost estimator is responsible for (a) implementing our cost model and (b) updating the statistics in the dictionary D. The plan generator probes the cost estimator for computing the costs of tasks involved in the plan according to the cost model described in Section III-C3.

The physical implementation of a logical operator's task in D is accompanied by a cost estimate. This stems from a known cost formula for the operator, parameterized by the input data size. The developer who registers an operator implementation may provide its cost formula. In several cases, though, the cost formula is unknown. To deal with this, HYPPO maintains standard, averaged statistics for operator execution collected by the monitor. Gradually, HYPPO learns from past pipeline runs and builds a cost model for pipeline operators based on crude estimate buckets rather than specific values. Operators can be profiled using performance models built using Bayesian Optimization, a popular method for the optimization of black-box functions that achieves reasonably accurate results with only a few test runs; a technique inspired by CherryPick [53]. Other methods could be used here, too.

TABLE I: Description of pipelines used from two Kaggle competitions. T is the number of teams participating in the competition, and S is the shape (rows, columns) of the original dataset.

Usecase	Description	T	S
HIGGS [50]	A competition that uses data from ATLAS experiment to identify the Higgs boson, the participants use different preprocessing techniques like Imputation, Scalar, and Polynomial features and for learning SVM is used with different regularization values similar to the pipelines used by [51].	1784	(800000, 30)
TAXI [52]	A competition to build a model that predicts the total ride duration of taxi trips in New York City. The primary dataset is released by the NYC Taxi and Limousine Commission, which includes pickup time, geo-coordinates, #passengers, and other variables. This use case involves more preprocessing steps, and a variety of regressor models are used.	1254	(1000000,11)

H. History Manager

The history manager (a) keeps track of the execution of new pipelines, maintains the history H accordingly, and (b) solves Problem 2. The first function involves updating H with new tasks that have been executed and new artifacts that have been generated. As new artifacts are being produced, it also performs the second function, which involves a critical decision: given a storage budget, which artifacts (from both, those already materialized and the newly created with the execution of a plan) to materialize for reducing the computation cost, and thus, the execution time, of future pipelines. The outcome of this decision is that certain artifacts that have been materialized will be evicted from storage, while some new artifacts will be stored. In the history H hypergraph, a materialized artifact v is represented as a hyperedge from source s to v, which is of type 'load' and whose cost represents the loading cost of retrieving the artifact from storage. Therefore, when evicting a materialized artifact, its corresponding 'load' hyperedge is removed—the node representing the artifact and all other hyperedges in its backward and forward star are kept. Note that data sources are not considered as candidates to be evicted.

In general, solving Problem 2 is an expensive mixed-integer linear program. For practical purposes, we opted for a greedy approach: we pick the artifact with the largest potential gain weighted by the plan locality coefficient, i.e., $\mathsf{pl}(v) \times \mathsf{gain}(v)$, as long as it fits in budget B. Then, we proceed iteratively until either no benefit to additional materialization is possible or all available storage has been used.

V. Experimental Evaluation

A. Setup

a) Software: HYPPO [54] has been developed in Python 3.10 (parser) and Java 8 (optimizer), uses NetworkX (3.1) for the parser, extends sklearn.pipelines (1.2.2) for the pipeline generator, and supports physical implementations from popular ML libraries including sklearn, Tensorflow 2.12, pyglmnet 1.1, lightgbm 3.3.5, libsvm 3.23, numpy 1.23.5, and scipy 1.10.1.

b) Datasets and pipelines: We consider two popular use cases from Kaggle competitions: HIGGS [50] and TAXI [52] (see also Table I). The same use cases were used and in the related literature as well [5].

We developed a pipeline generator that creates sequences of pipelines containing operators for preprocessing, learning, and evaluation for each use case. Our choice of logical operators was based on an analysis of popular submissions in the respective competitions, including classification (HIGGS) and regression (TAXI). We also used popular operators suggested in two comprehensive studies: (a) a survey of research papers from various domains [47], and (b) an analysis of millions of GitHub repositories and enterprise ML pipelines [49]. For each logical operator, we added physical implementations utilizing code submitted in the two competitions and using popular libraries [49]. We added a single implementation for use case specific preprocessing and evaluation operators, and at least two implementations for the rest.

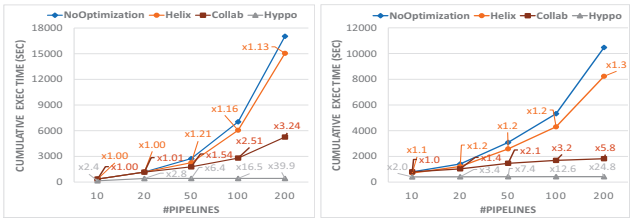
We created 5 sequences for each experiment, ran each experiment 5 times, and report the avg numbers for these runs.

c) Methods: We study the following methods that employ optimization strategies such as reuse, materialization, and equivalence. NoOptimization: Our straw man, a baseline approach that executes pipelines without any optimization. Sharing: Our second baseline enhances efficiency by identifying and eliminating common subexpressions across computations (i.e., reuse). Helix [4]: Considers computation sharing, employs materialization, and utilizes an optimization algorithm searching for the optimal solution (best possible plan based on the cost model employed). But Helix has two limiting assumptions: subsequent pipelines do not differ much (only one operator change is considered at each iteration), and does not keep history beyond the previous iteration. Collab [5]: Considers computation sharing, employs materialization, and its heuristic-based optimization strategy opts for speed, resulting in 'good enough plans'. HYPPO: Our proposed method considers computations sharing and enables reuse, materialization, and equivalence optimization strategies.

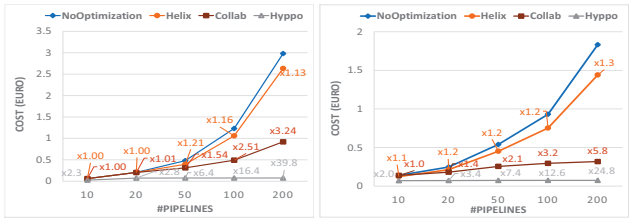
Although Helix and Collab support the same optimization strategies, they implement them differently, and so they produce different results. Collab's materialization strategy has been reported as more efficient than Helix's counterpart [5]. Our experiments corroborate this result. We noticed that Collab outperforms Helix due to its efficient materialization policy. Hence, we report Helix results only in our first experiment to avoid overcrowding the rest of the figures. HYPPO supports reuse, materialization, and equivalence. Hence, it is a comprehensive solution that draws from the strengths of both Helix and Collab while addressing their respective limitations.

B. Performance and cost analysis

1) (Scenario 1) Iterative pipeline execution: We consider a sequence of pipelines of length *#pipelines* and a *storage_budget* B that remains constant during pipeline execution. We start with an empty History H. We execute pipelines sequentially, and after each execution, we materialize artifacts until the storage budget is exhausted. After that, we replace artifacts according to each method's materialization strategy.

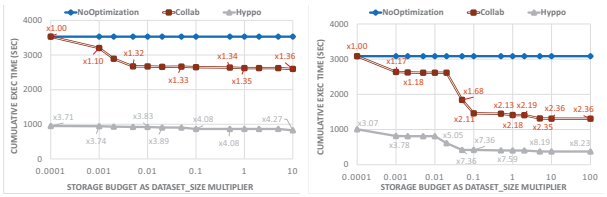


(a) HIGGS (b) TAXI

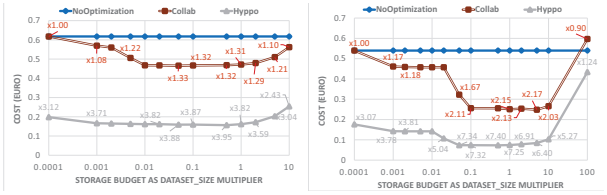


(c) HIGGS (d) TAXI

Fig. 3: Exec. time and price (cost) with varying #pipelines

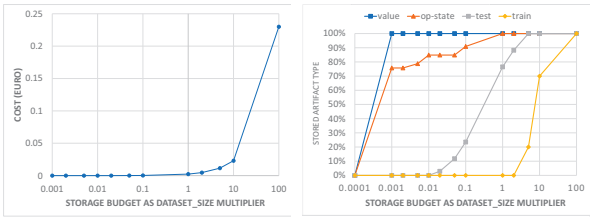


(a) HIGGS (b) TAXI

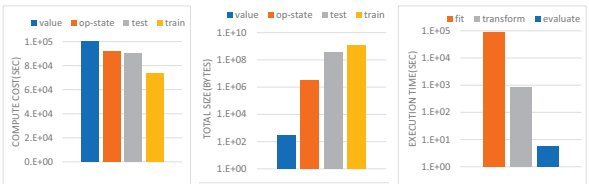


(c) HIGGS (d) TAXI

Fig. 4: Exec. time and price (cost) with varying storage budget

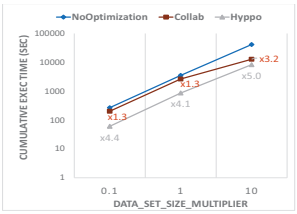


(a) storage cost (b) stored artifact types

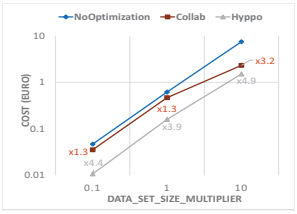


(c) artifact type cost (d) artifact type size (e) task type cost

Fig. 5: Study of artifact and task types



(a) HIGGS (b) TAXI



(c) HIGGS (d) TAXI

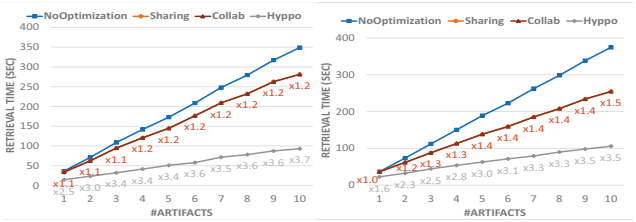
Fig. 6: Exec. time and price (cost) with varying dataset size

We monitor the following metrics: (a) *Cumulative Execution Time* (cet), measured in seconds, which captures the total duration required to run all pipelines; and (b) *Price*, measured in Euros, a derived metric computed as a function of cet and B as explained in Section III-C3. The variables price_per_time_unit and price_per_size_unit are derived as follows: we averaged the costs of deploying a computational instance (equivalent in capacity to our setup) across three cloud service providers: AWS [55], Google Cloud [56], and Azure [57]. Hence, the formula for calculating *price* is as follows: $\text{price} = \text{cet} \times 0.00018 + \text{B} \times 0.023$.

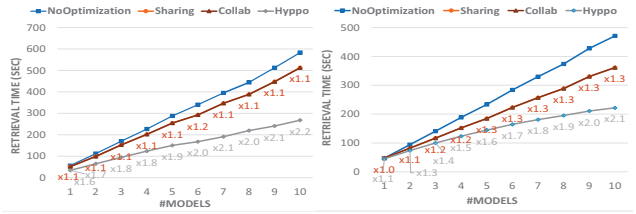
We present experiments with varying *#pipelines*, *storage budget*, and *dataset size*. We omit Sharing here as it resembles NoOptimization due to sequential pipeline execution.

a) Varying #pipelines: Figure 3 shows the time to finish all pipelines (cumulative execution time) with varying #pipelines, along with the corresponding price (cost in Euros). The values in the plot indicate a method's speedup compared to the baseline NoOptimization. For the first 10 pipelines, most tasks and artifacts are new (i.e., they have not been recorded yet in H). At this stage, Collab and Helix do not reap any benefits (speed-up 1×), indicating that little or no materialized artifacts are reused. HYPPO, however, achieves a 50% decrease in execution time, even though most of the artifacts and tasks are new due to its ability to handle equivalent expressions. As #pipelines increase, the recorded information in H grows; hence, the new pipelines executed contain a mix of new and known tasks and artifacts. Now, the benefits of Collab and Helix become apparent. Still, Collab performs better (up to 3.24× on HIGGS and 5.28× on TAXI) as it benefits from materialized artifacts, while Helix benefits mainly from reuse (up to 1.13× and 1.3×). HYPPO benefits from its combination of optimization strategies, and its gain almost doubles with the #pipelines, up to 40× on HIGGS and 25× on TAXI. Note that the HIGSS dataset is three times larger than the TAXI dataset, having 3× #features, and hence, the pipelines take longer to execute. Similar speed-ups exist for price gain, too (note that B is fixed in this experiment).

b) Varying storage budget: Figure 4 shows execution time and price gain speed-ups with varying values of storage

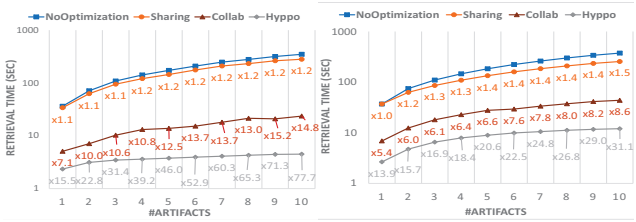


(a) HIGGS (b) TAXI

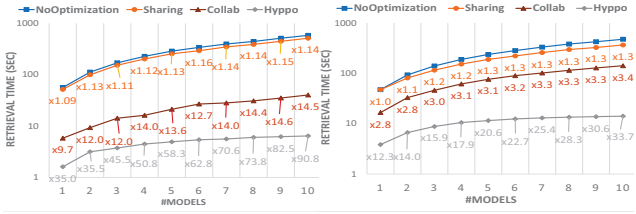


(c) HIGGS (d) TAXI

Fig. 7: Retrieval of artifacts and models, B=0



(a) HIGGS (b) TAXI



(c) HIGGS (d) TAXI

Fig. 8: Retrieval of artifacts and models, B=0.1

budget B as factors of the dataset size. Hence, B = 0.01 corresponds to a storage budget equal to 1% of the total dataset size. Note that for the same sequence of pipelines, B is fixed. For this experiment, we have #pipelines = 50. As we increase the storage budget, Collab achieves a performance boost up to 1.36x when the storage budget equals 10% of the total dataset size. This, however, comes at a significant price cost. On the other hand, HYPPO performs much better. For HIGGS, we observe that increasing the storage budget above 0.1 × dataset_size provides small benefits in terms of execution time, but it increases the price. In other words, storing more artifacts provides no benefit, which indicates that (re)computing these artifacts is more beneficial than loading them. For TAXI, we observe that increasing the storage budget above 0.1 × dataset_size provides small benefits in terms of time, which, however, are not reflected in terms of price. Thus, although loading artifacts improves execution time, still this reduction in time is not substantial enough to account for their storage cost. Artifacts produced by ML pipelines may have sizes in the same order of magnitude as the original dataset; our study clearly shows that storing artifacts comes at a cost.

c) Materialization decisions & beneficial artifact types: Figure 5(a) shows the monetary storage cost per budget, which when added to the monetary execution cost (proportional to Figures 4(a) and (b)) explains the trade-offs seen in the total monetary cost (price) in Figures 4(c) and (d). We delve deeper into the materializer's decisions. Figure 5(b) illustrates the percentage of stored artifacts by type when varying the storage budget, reflecting the different decisions made by the materializer as the storage budget increases. The materializer prioritizes artifacts of type *value* and then *op-state* as at 0.1 × dataset_size stores 100% of *value* artifacts and 90% of *op-state* artifacts. The next candidates in line are *test* and *train* artifacts. Figure 5(c) shows the average computational cost for each artifact type. Since every artifact type can be found at any point in the pipeline, they have similar computational costs, except for *value* type artifacts that are typically placed towards the end of a pipeline. Hence, they incur higher compute cost. Figure 5(d) shows the storage requirements of each artifact type: *value* (∼Bytes), *op-state* artifacts' size (∼KBytes), and lastly *test* and *train* (∼MBytes), which are in the same order as the original raw dataset; *train* is 3× the size of *test* as 3:1 is the split ratio between test and train set. Figure 5(e) shows that the execution time of *fit* tasks is 2 and 4 orders of magnitude higher than *transform* and *evaluate* tasks, respectively. As *op-state* are produced by *fit* tasks and can be up to 100× cheaper than data (*test* or *train*), it is usually better to store the *op-state*.

d) Varying dataset size: Figure 6 reports on the time and price gain speed-ups with varying size of the raw datasets. The data_set_multiplier indicates the dataset size used for the experiment with storage budget fixed at 0.1 × dataset_size and #pipelines = 50. Collab presents improved time and price gain speed-ups for larger datasets. However, HYPPO achieves much better speed-ups ranging from 4.3x up to 8.7x in both execution time and price gain, even for just 50 pipelines. With more pipelines, HYPPO performance will only improve as there will be additional optimization opportunities (e.g., sharing, stored artifacts). Hence, HYPPO shows excellent scalability with respect to pipeline load and dataset size.

2) (Scenario 2) Artifact and model retrieval: The first scenario assumes a cold start, where history is built with the pipeline sequence. Here we consider the steady mode of the system. That is, we start with a history created by 50 pipelines for each use case, HIGGS and TAXI. This scenario resembles several practical cases; e.g., users re-evaluate, visualise, and compare previously computed results. We investigate the time to retrieve: (a) a set of any artifact type and (b) fitted models. We investigate the behavior of the methods tested when the storage_budget B is *zero* and *0.1*. When B=0, the materialization strategy operates as Sharing. For the experiments, we generated five history graphs by executing five different sequences of 50 pipelines for each use case. We made 1000 random requests of different sizes in each graph and reported the average over all five graphs per use case per request size.

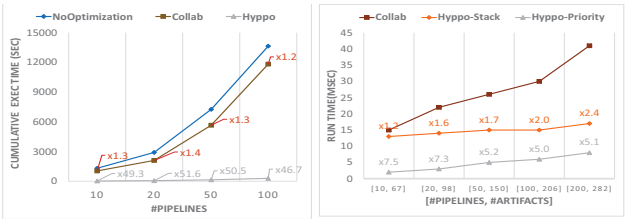


Fig. 9: (a) Advanced analysis, and (b) Optimization overhead

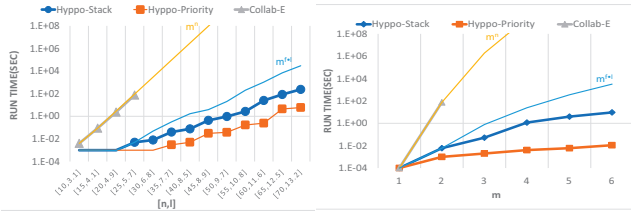


Fig. 10: Optimization runtime varying (a) n, and (b) m

a) Zero storage: Figure 7 shows the retrieval time of requesting a varying number of artifacts and models for HIGGS and TAXI (50 pipelines, B=0). Zero storage disables materialization; thus, the plots depict the benefit of considering equivalent options when building a plan. Without materialization, Collab performs as Sharing, showing a speed-up up to 1.2x for HIGGS and 1.5x for TAXI. HYPPO offers superior performance, from 3.7x to 3.5x for HIGGS and TAXI, respectively. Training models have a higher overall computation time than random artifacts. As this time is not shared, the benefits from sharing computations decrease when we request models. Yet, HYPPO achieves a 50% retrieval time even when models are requested. Note that requesting one artifact calls for the discovery of the best execution plan. Leveraging alternative plans, HYPPO achieves up to 3.7x speed-up.

b) 0.1 *storage:* Figure 8 shows the retrieval time of requesting a varying number of artifacts and models for HIGGS and TAXI. Here, we use 50 pipelines and B=0.1. Since B>0, materialization is enabled for both Collab and HYPPO and offers them both significant speed-ups. The benefits here come primarily from the artifacts/models stored. HYPPO and Collab store 83% and 55% (HIGGS) and 64% and 54% (TAXI) of the total artifacts, respectively. HYPPO effectiveness is due to its ability to identify equivalent artifacts and make better use of B. Note that HYPPO achieves larger coverage in HIGGS as this dataset is 3x bigger than TAXI, so its B (10% of the dataset size) is larger too.

3) (Scenario 3) Advanced analysis: Many users in the TAXI competition [52] employ model ensemble learning operators, which use trained models and extend previous pipelines. Here, we start with a History created by 100 TAXI pipelines and generated additional workloads of various #pipelines (from 10 to 100) that use models trained in the past, similar to the ones found in [52], that leverage ensemble operators such as StackingRegressor and VotingRegressor. Figure 9(a) shows that in this scenario, HYPPO achieves a speed-up up to 50x, whilst Collab has a moderate speed-up of up to 1.4x.

4) (Scenario 4) Optimization overhead: Figure 9(b) reports on the execution time of the optimization methods for various $\langle$#pipelines, #Hnodes$\rangle$ pairs. We observe that HYPPO runtime scales gracefully with increasing the complexity of the search space (History size). On the other hand, Collab's runtime increases rapidly with more complex configurations, which questions its value in practical scenarios.

5) Scalability: To investigate the scalability of HYPPO in finding the optimal plan, we developed a synthetic hypergraph generator, which has two parameters: #artifacts (n) and #alternatives per artifact (m). We start by generating pipelines, akin to our two usecases, until the node count reaches n. Then, we introduce additional edges to nodes to satisfy degree m. Nodes lacking outgoing edges are designated as request targets T. We assess the performance of our algorithms, HYPPO-STACK and HYPPO-PRIORITY, against COLLAB-E, an approach that generates a DAG for each combination of alternatives, and executes the Collab reuse algorithm for each of them. COLLAB-E, in contrast to COLLAB, finds the optimal plan under equivalences, as do the HYPPO variants.

Figure 10(a) presents the runtime of the methods across various graph sizes, denoted as $[\mathsf{n}, \bar{\ell}]$ pairs, where $\bar{\ell}$ is the average maximum path length, while keeping the number of alternatives at $\mathsf{m} = 2$. (We report the average of 10 executions.) The exhaustive nature of COLLAB-E makes it impractical, as it takes more than an hour to find the best plan in a graph of 30 artifacts with a length not greater than 7. In contrast, the HYPPO variants require less than a few minutes even for big pipelines of length up to 14, with HYPPO-PRIORITY being more scalable. In the plot, we also draw the complexity functions $\mathsf{O}(\mathsf{m}^{\mathsf{n}})$, which represents exhaustive search, and $\mathsf{O}(\mathsf{m}^{\mathsf{f}\cdot\ell})$, which concerns the OPTIMIZE algorithm; we anchor these functions at the first value of COLLAB-E and HYPPO-STACK, respectively. Interestingly, all methods scale roughly as their theoretical complexity predicts.

In Figure 10(b), we vary the number m of alternatives while keeping the number of artifacts fixed at the largest value n=4 where COLLAB-E executes within one hour. COLLAB-E does not scale well, while HYPPO presents excellent scalability.

VI. CONCLUSION

This work presented HYPPO, a system that exploits equivalences among artifacts and tasks to reveal reuse opportunities beyond materialization from historical ML pipeline executions. Given an ML pipeline, HYPPO returns an optimized execution plan. It uses a hypergraph representation of pipelines and the history to (a) search for an optimized execution plan, and (b) decide what artifacts to materialize. An evaluation on a variety of use-cases shows that HYPPO finds optimized execution plans that are up to two orders of magnitude faster and cheaper than the original pipeline.

Acknowledgments: This work has been partially supported by the H2020-MSCAITN-2020 DEDS (GA.955895), the EU-HORIZON programmes FAIR-CORE4EOSC (GA.101057264), CREXDATA (GA.101092749), CyclOps (GA.101135513), and the Spanish Ministerio de Ciencia e Innovación under project PID2020-117191RB-I00/AEI/10.13039/501100011033 (DOGO4ML).

REFERENCES

[1] A. Carqueja, B. Cabral, J. P. Fernandes, and N. Lourenço, "On the democratization of machine learning pipelines," in *2022 IEEE Symposium Series on Computational Intelligence (SSCI)*, 2022, pp. 455–462.

[2] comet, "ML industry report — 2021 machine learning practitioner survey." [Online]. Available: https://go.comet.ml/report-machine-learning-practitioners-survey.html

[3] X. He, K. Zhao, and X. Chu, "Automl: A survey of the state-of-the-art," *Knowl. Based Syst.*, vol. 212, p. 106622, 2021.

[4] D. Xin, S. Macke, L. Ma, J. Liu, S. Song, and A. Parameswaran, "HELIX: Holistic optimization for accelerating iterative machine learning," vol. 12, no. 4, p. 446–460, dec 2018. [Online]. Available: https://doi.org/10.14778/3297753.3297763

[5] B. Derakhshan, A. Rezaei Mahdiraji, Z. Abedjan, T. Rabl, and V. Markl, "Optimizing machine learning workloads in collaborative environments," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1701–1716. [Online]. Available: https://doi.org/10.1145/3318464.3389715

[6] A. Phani, B. Rath, and M. Boehm, "LIMA: Fine-grained lineage tracing and reuse in machine learning systems," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1426–1439. [Online]. Available: https://doi.org/10.1145/3448016.3452788

[7] C. Zhang, A. Kumar, and C. Ré, "Materialization optimizations for feature selection workloads," *ACM Trans. Database Syst.*, vol. 41, no. 1, feb 2016. [Online]. Available: https://doi.org/10.1145/2877204

[8] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal, "Optimizing analytic data flows for multiple execution engines," ser. SIGMOD '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 829–840. [Online]. Available: https://doi-org.recursos.biblioteca.upc.edu/10.1145/2213836.2213963

[9] N. Giatrakos, D. Arnu, T. Bitsakis, A. Deligiannakis, M. Garofalakis, R. Klinkenberg, A. Konidaris, A. Kontaxakis, Y. Kotidis, V. Samoladas, A. Simitsis, G. Stamatakis, F. Temme, M. Torok, E. Yaqub, A. Montagud, M. Ponce de León, H. Arndt, and S. Burkard, "INforE: Interactive cross-platform analytics for everyone," in *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, ser. CIKM '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 3389–3392. [Online]. Available: https://doi.org/10.1145/3340531.3417435

[10] S. Alsudais, A. Kumar, and C. Li, "Raven: Accelerating execution of iterative data analytics by reusing results of previous equivalent versions," in *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, ser. HILDA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3597465.3605219

[11] A. G. Georgia Kougka and A. Simitsis, "The many faces of data-centric workflow optimization: a survey." Springer Science and Business Media LLC, 2018. [Online]. Available: https://doi.org/10.1007/s41060-018-0107-0

[12] Kubeflow, "The machine learning toolkit for kubernetes," 2023, available at: https://www.kubeflow.org.

[13] M. Boehm, I. Antonov, S. Baunsgaard, M. Dokter, R. Ginthör, K. Innerebner, F. Klezin, S. N. Lindstaedt, A. Phani, B. Rath, B. Reinwald, S. Siddiqui, and S. B. Wrede, "SystemDS: A declarative machine learning system for the end-to-end data science lifecycle," in *CIDR*, 2020.

[14] AzureML, "Azure machine learning: Machine-learning-as-a-service," 2023, available at: https://azure.microsoft.com/de-de/services/machine-learning.

[15] MLflow, "An open source platform for the machine learning lifecycle," 2023, available at: https://mlflow.org.

[16] M. Schlegel and K.-U. Sattler, "Management of machine learning lifecycle artifacts: A survey," *SIGMOD Rec.*, vol. 51, no. 4, p. 18–35, jan 2023. [Online]. Available: https://doi.org/10.1145/3582302.3582306

[17] S. Idowu, D. Strüber, and T. Berger, "Asset management in machine learning: A survey," in *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '21. IEEE Press, 2021, p. 51–60. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP52600.2021.00014

[18] Z. Shang, E. Zgraggen, B. Buratti, F. Kossmann, P. Eichmann, Y. Chung, C. Binnig, E. Upfal, and T. Kraska, "Democratizing data science through interactive curation of ml pipelines," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD. New York, NY, USA: Association for Computing Machinery, 2019, p. 1171–1188. [Online]. Available: https://doi.org/10.1145/3299869.3319863

[19] D. Patel, S. Shrivastava, W. Gifford, S. Siegel, J. Kalagnanam, and C. Reddy, "Smart-ml: A system for machine learning model exploration using pipeline graph," in *2020 IEEE International Conference on Big Data (Big Data)*, 2020, pp. 1604–1613.

[20] M. J. Smith, C. Sala, J. M. Kanter, and K. Veeramachaneni, "The machine learning bazaar: Harnessing the ml ecosystem for effective system development," ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 785–800. [Online]. Available: https://doi.org/10.1145/3318464.3386146

[21] J.-W. Cui, W. Lu, X. Zhao, and X.-Y. Du, "Efficient model store and reuse in an OLML database system," *Journal of Computer Science and Technology*, vol. 36, pp. 792–805, 07 2021.

[22] M. Zhao, L. Chen, K. Yang, Y. Du, and Y. Gao, "Finding materialized models for model reuse," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–16, 2023.

[23] M. Vartak, H. Subramanyam, W.-E. Lee, S. Viswanathan, S. Husnoo, S. Madden, and M. Zaharia, "ModelDB: A system for machine learning model management," in *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, ser. HILDA '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2939502.2939516

[24] M. Vartak, J. M. F. da Trindade, S. Madden, and M. Zaharia, "MISTIQUE: A system to store and query model intermediates for model diagnosis," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1285–1300. [Online]. Available: https://doi.org/10.1145/3183713.3196934

[25] J. M. Kleinberg and É. Tardos, *Algorithm design*. Addison-Wesley, 2006.

[26] G. Graefe and W. McKenna, "The volcano optimizer generator: extensibility and efficient search," in *Proceedings of IEEE 9th International Conference on Data Engineering*, 1993, pp. 209–218.

[27] S. Cohen, W. Nutt, and A. Serebrenik, "Rewriting aggregate queries using views," in *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '99. New York, NY, USA: Association for Computing Machinery, 1999, p. 155–166. [Online]. Available: https://doi.org/10.1145/303976.303992

[28] Z. Xu, G. T. Kakkar, J. Arulraj, and U. Ramachandran, "EVA: A symbolic approach to accelerating exploratory video analytics with materialized views," in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 602–616. [Online]. Available: https://doi.org/10.1145/3514221.3526142

[29] L. Ramjit, M. Interlandi, E. Wu, and R. Netravali, "Acorn: Aggressive result caching in distributed data processing frameworks," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 206–219. [Online]. Available: https://doi.org/10.1145/3357223.3362702

[30] J. LeFevre, J. Sankaranarayanan, H. Hacigumus, J. Tatemura, N. Polyzotis, and M. J. Carey, "Opportunistic physical design for big data analytics," ser. SIGMOD '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 851–862. [Online]. Available: https://doi.org/10.1145/2588555.2610512

[31] Y. Foufoulas and A. Simitsis, "Efficient execution of user-defined functions in SQL queries," *Proc. VLDB Endow.*, vol. 16, no. 12, pp. 3874–3877, 2023.

[32] Y. E. Foufoulas, A. Simitsis, E. Stamatogiannakis, and Y. E. Ioannidis, "YeSQL: "you extend SQL" with rich and highly performant user-defined functions in relational databases," *Proc. VLDB Endow.*, vol. 15, no. 10, pp. 2270–2283, 2022.

[33] P. Jovanovic, A. Simitsis, and K. Wilkinson, "Engine independence for logical analytic flows," in *2014 IEEE 30th International Conference on Data Engineering*, 2014, pp. 1060–1071.

[34] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht, "KeystoneML: Optimizing pipelines for large-scale advanced analytics," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, 2017, pp. 535–546.

[35] W. McKinney, "Data structures for statistical computing in python," in *SciPy 2010*, 2010, pp. 56–61.

[36] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[37] C. Berge, *Graphs and Hypergraphs*, ser. Graphs and Hypergraphs, 1973.

[38] G. Ausiello, A. D'Atri, and D. Saccà, "Graph algorithms for functional dependency manipulation," *J. ACM*, vol. 30, no. 4, pp. 752–766, 1983.

[39] G. Gallo, G. Longo, and S. Pallottino, "Directed hypergraphs and applications," *Discret. Appl. Math.*, vol. 42, pp. 177–201, 1993. [Online]. Available: https://doi.org/10.1016/0166-218X(93)90045-P

[40] A. Rheinländer, A. Heise, F. Hueske, U. Leser, and F. Naumann, "SOFA: an extensible logical optimizer for udf-heavy data flows," *Inf. Syst.*, vol. 52, pp. 96–125, 2015. [Online]. Available: https://doi.org/10.1016/j.is.2015.04.002

[41] P. Jovanovic, O. Romero, A. Simitsis, A. Abelló, and D. Mayorova, "A requirement-driven approach to the design and evolution of data warehouses," *Inf. Syst.*, vol. 44, pp. 94–119, 2014.

[42] P. Jovanovic, O. Romero, A. Simitsis, and A. Abelló, "Incremental consolidation of data-intensive multi-flows," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 5, pp. 1203–1216, 2016.

[43] A. Simitsis, P. Vassiliadis, and T. K. Sellis, "Optimizing ETL processes in data warehouses," in *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*. IEEE Computer Society, 2005, pp. 564–575.

[44] A. Simitsis, P. Vassiliadis, M. Terrovitis, and S. Skiadopoulos, "Graph-based modeling of ETL activities with multi-level transformations and updates," in *DaWaK*, ser. Lecture Notes in Computer Science, vol. 3589. Springer, 2005, pp. 43–52.

[45] R. T. Wong, "A dual ascent approach for steiner tree problems on a directed graph," *Math. Program.*, vol. 28, no. 3, pp. 271–287, 1984. [Online]. Available: https://doi.org/10.1007/BF02612335

[46] Y. Kotidis and N. Roussopoulos, "Dynamat: A dynamic view management system for data warehouses," in *SIGMOD*, 1999, pp. 371–382.

[47] D. Xin, L. Ma, S. Song, and A. G. Parameswaran, "How developers iterate on machine learning workflows - A survey of the applied machine learning literature," *CoRR*, vol. abs/1803.10311, 2018.

[48] A. Simitsis, K. Wilkinson, U. Dayal, and M. Hsu, "HFMS: managing the lifecycle and complexity of hybrid analytic data flows," in *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. IEEE Computer Society, 2013, pp. 1174–1185.

[49] F. Psallidas, Y. Zhu, B. Karlas, J. Henkel, M. Interlandi, S. Krishnan, B. Kroth, V. Emani, W. Wu, C. Zhang, M. Weimer, A. Floratou, C. Curino, and K. Karanasos, "Data science through the looking glass: Analysis of millions of github notebooks and ML.NET pipelines," *SIGMOD Rec.*, vol. 51, no. 2, p. 30–37, jul 2022. [Online]. Available: https://doi.org/10.1145/3552490.3552496

[50] Kaggle, "Higgs boson machine learning challenge," 2014. [Online]. Available: https://www.kaggle.com/competitions/higgs-boson/overview

[51] B. Derakhshan, A. Rezaei Mahdiraji, Z. Kaoudi, T. Rabl, and V. Markl, "Materialization and reuse optimizations for production data science pipelines," in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1962–1976. [Online]. Available: https://doi.org/10.1145/3514221.3526186

[52] Kaggle, "New york city taxi trip duration," 2017. [Online]. Available: https://www.kaggle.com/competitions/nyc-taxi-trip-duration/overview

[53] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "CherryPick: Adaptively unearthing the best cloud configurations for big data analytics," in *USENIX NSDI 17*, 2017, pp. 469–482.

[54] HYPPO, "Code repository," 2024, available at: https://github.com/akontaxakis/HYPPO.

[55] AWS, "Amazon web services," 2023, available at: https://aws.amazon.com.

[56] Google Cloud, "Google cloud: Cloud computing services," 2023, available at: https://cloud.google.com.

[57] Azure, "Microsoft azure: Cloud computing services," 2023, available at: https://azure.microsoft.com.