Materialization and Reuse Optimizations for Production Data Science Pipelines

Behrouz Derakhshan
behrouz.derakhshan@gmail.com
DFKI GmbH
Berlin, Germany

Alireza Rezaei Mahdiraji
alireza.mahdiraji@yara.com
Yara Digital Production
Berlin, Germany

Zoi Kaoudi
zoi.kaoudi@tu-berlin.de
TU Berlin
Berlin, Germany

Tilmann Rabl
tilmann.rabl@hpi.de
HPI, University of Potsdam
Potsdam, Germany

Volker Markl
volker.markl@tu-berlin.de
DFKI GmbH, TU Berlin
Berlin, Germany

ABSTRACT

Many companies and businesses train and deploy machine learning (ML) pipelines to answer prediction queries. In many applications, new training data continuously becomes available. A typical approach to ensure that ML models are up-to-date is to retrain the ML pipelines following a schedule, e.g., every day on the last seven days of data. Several use cases, such as A/B testing and ensemble learning, require many pipelines to be deployed in parallel. Existing solutions train each pipeline separately, which generates redundant data processing. Our goal is to eliminate redundant data processing in such scenarios using materialization and reuse optimizations. Our solution comprises of two main parts. First, we propose a materialization algorithm that given a storage budget, materializes the subset of the artifacts to minimize the run time of the subsequent executions. Second, we design a reuse algorithm to generate an execution plan by combining the pipelines into a directed acyclic graph (DAG) and reusing the materialized artifacts when appropriate. Our experiments show that our solution can reduce the training time by up to an order of magnitude for different deployment scenarios.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; • **Information systems** → **Database management system engines**.

KEYWORDS

machine Learning pipeline, materialization, reuse

ACM Reference Format:
Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Zoi Kaoudi, Tilmann Rabl, and Volker Markl. 2022. Materialization and Reuse Optimizations for Production Data Science Pipelines. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3514221.3526186

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3526186

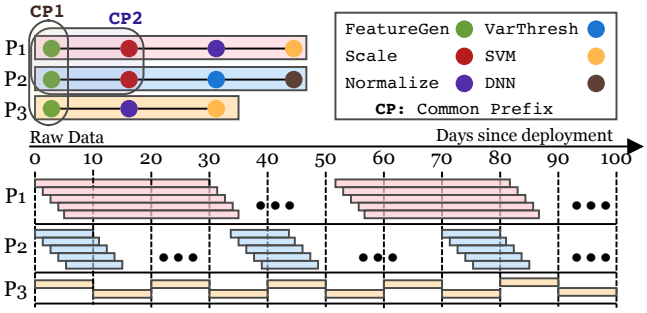


Figure 1: Execution of scheduled pipelines and the generated artifacts after every execution.

1 INTRODUCTION

Designing machine learning (ML) pipelines and training ML models are only the first steps in industrial settings. After the initial training, data scientists deploy the pipelines into a production environment to answer prediction queries. State-of-the-art solutions typically require *multiple* pipelines to maximize the prediction accuracy. Such practices are common in A/B testing [19, 31, 45], continuous integration of ML pipelines, where multiple pipelines are trained and tested before one is pushed to production [6, 30, 42], and automated or manual design and training of pipelines ensemble [8, 17].

In most enterprise applications, new training data arrives continuously. The current approach that ensures models stay up-to-date is to first utilize a lambda-style architecture [36] to store the continuously arriving data in a persistent storage unit. The stored data is partitioned by a time unit based on the timestamp of the data (e.g., hours or days). Then, the pipelines are retrained over past data intervals, e.g., a moving window over the last n days [3, 53]. Typically, the data scientists define the data interval and the frequency of the retraining. Retraining is paramount in online applications, such as ads click prediction [26] and mobile application recommendation [3], where the introduction of new products and users requires retraining of the pipelines to provide meaningful predictions.

Example Use Case. A mobile advertising company has a data collection system to store the served ads and their click outcome (clicked or not). The data arrives continuously, where each data point has a timestamp. The data collection system ingests the incoming data and stores the data partitioned by the time-unit day

of the timestamp. The data scientists build ML pipelines to train models that predict the likelihood of a click. They utilize ensemble learning and train three pipelines, i.e., p_1, p_2, and p_3, to combine their predictions. Figure 1 shows the executions of the pipelines in the first 100 days after the deployment. The pipelines consist of various components: feature generator (for generating polynomial and interaction features), standard scaler, normalizer, variance thresholding, and three ML models, i.e., SVM, DNN, and Logistic Regression (LR). CP represents a common prefix. For example, p_1 and p_2 have the common prefix `FeatureGen`→`Scale` (CP2) and all three pipelines have the common prefix `FeatureGen` (CP1).

The data scientists retrain p_1 and p_2 daily on the last 30 and 10 days of data, and p_3 every 10 days on the last 10 days of data. Every pipeline execution results in three categories of artifacts: (1) the computed statistics, such as the mean and variance for the scaler, (2) the transformed features after every component, and (3) the trained models after every execution. For example, after the first execution, the feature generator and normalize components of p_1 generate polynomial feature and normalized artifacts in the interval $[0, 30)$. Similarly, the scale component of p_1 generates mean and variance statistics artifacts and scaled feature artifacts in the interval $[0, 30)$.

Problem. Although several existing systems provide end-to-end management of ML pipelines [3, 7, 13], they only process one pipeline at-a-time. Therefore, in multi-pipeline scenarios, every pipeline is retrained in isolation. The pipelines in these scenarios are often very similar, and, thus, share many components. For example, data cleaning components [27], standardization and normalization of numerical data, and encoding of categorical data are standard techniques in the design of data science and ML pipelines. Furthermore, in many ensemble learning approaches, the data processing components of the pipelines are the same and only the ML models are different. In addition, current state-of-the-art solutions ignore the fact that it is the same group of pipelines that are retrained but in different intervals. Therefore, retraining pipelines in isolation results in redundant data processing, which leads to longer training times and, thus, delays in keeping ML models up-to-date.

Optimization Opportunities. By storing the generated artifacts (i.e., statistics, features, and models) in fast storage (e.g., RAM), we identify four possible opportunities to optimize the execution of multiple pipelines. (i) If we materialize the computed statistics, we can reuse them later to avoid redundant computation. For example, every execution of p_1 has a 29-day overlap with the previous execution. On the second execution (on the interval $[1, 31)$), since mean and variance can be computed incrementally, we reuse the mean and variance of the interval $[1, 30)$ and only compute the mean and variance of $[30, 31)$. (ii) Similar to statistics, if we materialize the intermediate features, we can reuse them in the subsequent executions. For example, the second execution of p_1 can reuse the generated features in the interval $[1, 30)$. (iii) By materializing the trained models, we can reuse them as initial points (warmstarting) for subsequent executions. (iv) Deployed pipelines may share similar components. Similar to multi-query optimization techniques [46], we can exploit common sub-expressions (pipeline prefixes in our case) to eliminate redundant data processing and loading of the materialized data. For example, if the result of the first two components of p_1 and p_2 are materialized, then, we can load them once and reuse them for both pipelines. Otherwise, p_1 and p_2 can share the execution of their first two components, when no artifacts are materialized. The first three opportunities exploit intra-pipeline, while the last one exploits inter-pipeline similarities.

Solution. Inspired by materialized view selection [35] and multi-query optimization (MQO) [9, 44, 46], we present a system for materializing and reusing generated artifacts to optimize the retraining of multiple deployed ML pipelines. We assume the input (training data) arrives continuously and is stored in a partitioned format based on a user-selected time unit. Each pipeline retrains on a moving window over the last n partitions. In contrast to existing works on materialization and MQO, our system tackles the following challenges: (i) *Heterogeneity of the generated artifacts*: Generated artifacts can be features, statistics, and ML models. This is in contrast to traditional view materialization and MQO techniques, where the operator output is only tuples. To enable materialization and reuse, we must determine an appropriate structure to represent multiple ML pipelines and unify their cost representation via a cost model. (ii) *Limited storage*: Depending on the dataset size and complexity of the ML pipelines, it may not be possible to materialize all the generated artifacts due to storage constraints. To maximize the reuse opportunities, we must associate the reduction in cost with the storage overhead. Our solution should also adapt to the changing workload characteristics, i.e., the pipelines and their schedules, as pipelines are added or removed. (iii) *Variety of execution plans*: Given the set of scheduled pipelines and existing materialized artifacts, there can be different ways of determining what (materialized) artifacts to load, what artifacts to compute, and when to merge artifacts. We must devise a reuse algorithm that finds the optimal (according to our cost model) execution plan. (iv) *Data distribution change*: The distribution of incoming data may change. Our system must handle such distribution changes to produce correct results and not reuse stale information.

Contributions. In this paper, we tackle the above challenges and make several contributions. We propose a system for optimizing the training of ML pipelines via materialization and reuse (Section 4). We formulate the problem of materialization and reuse of the artifacts in ML pipelines and devise a unified cost model for all types of artifacts. We also propose an algorithm that computes the materialization benefit of artifacts under limited storage (Section 5). We introduce a reuse algorithm that generates the optimal execution plan and handles distribution shift (Section 6). Finally, we conduct an experimental evaluation with multiple datasets and ML pipelines to show that our system improves performance by up to an order-of-magnitude compared to current practices (Section 7).

2 RELATED WORK

Multi-query optimization (MQO) and view materialization in database systems, which have the goal of optimizing the execution of multiple SQL queries, either by sharing the execution of common sub-expressions or finding a common set of views to materialize, have been studied for over 30 years [9, 35, 44, 46]. More recent works, such as Nectar [22], ReStore [15], MRShare [40], and the work of Wang et al. [55] utilize MQO and materialization and reuse of intermediate data in parallel execution frameworks such as MapReduce [11] and DryadLINQ [16]. Such works can be divided

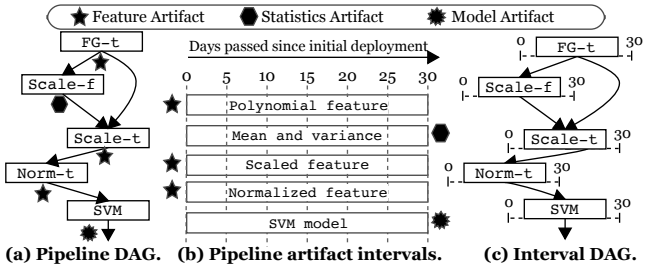


Figure 2: DAG, Interval DAG, and artifacts of a pipeline.

into three groups. The first group [15, 22] solely rely on materializing and reusing sub-expressions, e.g., the result of map or reduce functions, with no MQO. The second group only offers MQO by sharing the scan or map outputs [40]. And the third group offers MQO and materialization by both sharing scans and caching the result of map and reduce operators [55]. Some works on streaming data also utilize materialization and reuse to optimize the execution of queries [29, 32, 49, 51], however, they only share intermediate data for aggregation and join queries. Our solution differs from such approaches, as we focus on MQO and materialization and reuse to optimize the execution and retraining of ML pipelines on *overlapping intervals of time-ordered data*. In addition, our approach takes *heterogeneous intermediate results* into account (i.e., ML models, statistics, and features), in contrast to the previous approaches that deal only with tuples and cannot be trivially used in our setting.

Several existing works in the ML domain also utilize materialization and reuse to optimize the execution of workloads. Particularly, they offer materialization and reuse in workloads such as collaborative ML [12], iterative ML [56], model diagnosis [54], approximate ad-hoc exploratory ML [24, 25], interactive feature selection [59], and knowledge base construction [48]. In contrast to our work, they focus either on the execution of the data preprocessing phase or on the training of ML models. In addition, they do not consider the problem of optimizing the execution and retraining of ML pipelines on overlapping intervals of time-ordered data.

[53] can be considered close to our work as it proposes a system for versioning and reusing the intermediate data in ML pipelines. However, the authors propose a brute-force approach, i.e., materializing all the intermediates and execution of one pipeline at a time. Existing ML deployment systems [3, 13] also optimize the retraining of the deployed pipelines. However, such works only support the execution of one pipeline at a time and only materialize the last-level features. Furthermore, existing ML deployment systems do not address the problem of distribution change.

3 PRELIMINARIES

In this section, we define pipeline DAGs, the different artifact types generated after a pipeline execution, and how we reuse each type.

Pipeline DAG. Conventionally, a pipeline is a chain of operators. To provide support for the popular `fit/transform` API utilized in many data science tools (e.g., scikit-learn [41] and SparkML [38]), we convert the pipelines represented in such tools into Directed Acyclic Graphs (DAGs) and refer to these when we discuss data science pipelines. With such a DAG representation, our materialization and reuse algorithms can support not only unary operators, such as transformations, but also n-ary operators, such as aggregation, join, and union. The `fit` function, which computes some internal states (such as the mean and variance of a scaler), outputs the computed values. Figure 2a shows the DAG of pipeline p_1 of our example use case (Section 1). For the standard scaler, `Scale_f` and `Scale_t` show the `fit` and `transform` operators.

Pipeline Artifacts. Executing a pipeline over an interval generates an *artifact*. We define three different types of artifacts: (1) Intermediate transformed data, which we refer to as the feature artifacts, are the result of data transformations such as `transform`, `join`, or `union`. (2) Computed statistics are artifacts that result from the `fit` or other aggregation operators. (3) ML models are artifacts that are resulting from training a model on features. Figure 2b shows the generated artifacts after the first execution of p_1 of our example use case on the interval $[0, 30)$ days. Feature and statistics artifacts are splittable. As a result, we can slice the artifacts into sub-intervals. To ensure that statistics artifacts are splittable, we only support incrementally computable statistics (or those statistics with an approximate incremental version, e.g., approximate quantiles [21]), as well as, algebraic and distributive aggregations [20]. However, model artifacts are not splittable. For example, in Figure 2b, there is only one SVM model in the interval $[0, 30)$, whereas, other artifacts can be split into sub-intervals (dashed lines going through an artifact means the artifact can be split into sub-intervals).

Interval DAG. We utilize a modified version of the pipeline DAG, which we call *Interval DAG*. The vertices of the interval DAG encapsulate both the operators and the intervals that they operate on. Figure 2c shows the interval DAG of p_1 for its execution in the interval $[0, 30)$. We represent a vertex of the interval DAG by $v = \langle o, i \rangle$ where o is the operator and i is the interval (e.g., $\langle$`FG-t`$, [0, 30)\rangle$ in Figure 2c). The interval DAG has different uses. For example, our materialization algorithm uses an interval DAG and assigns a benefit score to every vertex and the reuse algorithm generates an execution plan in the form of an interval DAG.

Reuse Procedures. We define two reuse procedures. Feature and statistics artifacts reuse involves loading the materialized sub-interval of artifacts from storage instead of computing them (i.e., reusing an artifact replaces its computation). For example, for the second execution of p_1 in interval $[1, 31)$, we can reuse the statistics artifacts (i.e., mean and variance) and the feature artifacts of the interval $[1, 30)$. Model reuse is the process of using a materialized model to *warmstart* the training procedure with the parameters of the materialized model. Therefore, reusing a model does not replace its computation, but reduces the training time [1, 52].

4 SYSTEM ARCHITECTURE

In this section, we first present the system architecture. Then, we discuss its extensibility for enabling the implementation of the user-defined pipelines, new run time engines, and data stores.

4.1 System Components

Figure 3 shows the components and workflow of our system. Users can implement pipeline operators, design ML pipelines, and define schedules for them. Users can also add new scheduled pipelines,

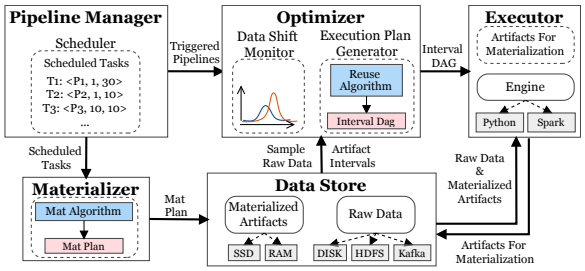


Figure 3: Workflow and architecture of the system.

which we refer to as a *task*, and modify/delete existing tasks. The rest of this section describes the different components of our system.

Pipeline Manager. The pipeline manager keeps track of the submitted tasks. Each task specifies the ML pipeline, the slide (the execution frequency), and the interval (the size of the training data interval). The pipeline manager interacts with two other components, i.e., the materializer and the optimizer. Once users add, modify, or delete a task, the pipeline manager informs the materializer to (re)create a materialization plan. Furthermore, the pipeline manager checks the schedule of the tasks and—based on the slide values—triggers their execution accordingly. Whenever a set of tasks are triggered, the pipeline manager transfers the pipelines and the interval of the training data to the optimizer.

Materializer. Given the scheduled tasks, the materializer constructs a materialization plan using our novel materialization algorithm (Section 5). The plan specifies the artifacts with their intervals that the system should materialize. For example, the plan can indicate that the last ten days of mean and variance statistics and the normalized features of p_1 (Figure 2) should be materialized.

Optimizer. The optimizer receives the triggered pipelines and their respective data intervals from the pipeline manager. The optimizer has two main responsibilities, i.e., detecting distribution shifts and generating the execution plan. When processing time-dependent data, it is vital to detect and handle distribution shifts. In such a scenario, reusing materialized artifacts may generate incorrect results. The data shift monitor is responsible for detecting distribution shifts. It analyzes a sample of the raw data to indicate the presence/absence of distribution shift. In our current implementation, we use the Kolmogorov-Smirnov test [14], but other techniques can be implemented as well. In case of a distribution shift, the data shift monitor notifies the execution plan generator to invalidate any affected materialized artifacts.

To generate the execution plan, the optimizer queries the data store for materialized artifacts. Given the materialized artifacts, the optimizer utilizes our reuse algorithm (Section 6) to generate an execution plan with the smallest cost based on our cost model. The execution plan is an interval DAG where vertices specify the interval of the data for processing and whether the data is materialized.

Executor. The executor component receives the interval DAG from the optimizer and visits the vertices of the interval DAG in topological order. For a materialized vertex, it loads the artifact from the data store. For an unmaterialized vertex, the executor runs the operator. After the execution, the executor sends the generated artifacts to the data store, which then decides what artifacts to materialize based on the materialization plan. The engine inside the executor is responsible for the actual execution. The engine is extensible and system users can implement new run times. Currently, we implement a Python and a Spark [58] execution engine.

Data Store. The data store component is responsible for the storage and management of the incoming raw training data and the materialized artifacts. It keeps the materialization plan constructed by the materializer and requests the executor to return the artifacts that are part of the plan. The data store also provides a sample of the raw data to the optimizer for shift detection. Furthermore, during the plan generation, the data store informs the optimizer of the interval of the materialized artifacts.

The data store can interface with different underlying storage systems (e.g., HDFS). In our system, new data continuously arrives and is partitioned based on a user-selected time unit (e.g., hourly or daily). The data store uses RAM for storing the materialized artifacts since we require fast access to them. However, the data store is agnostic to the underlying storage. Other fast storage types, such as SSD, can replace RAM for storing materialized artifacts.

4.2 Extensibility

We design the system to be extensible. In general, there are two aspects of extensibility that we consider, i.e., new run times and new pipelines with different operators.

Engine and Data Store Extension. Currently, we provide support for Python-based and Spark-based engines. Support for a new engine requires two steps. First, the user needs to implement the physical operators that are included in the physical plans in the new engine. Second, each engine may require the raw and the materialized artifacts to be in a specific format. Currently, we have a Numpy-based [23] data store for the Python engine and an RDD-based [57] data store for the Spark engine. Engines that require other types of data need a new data store as well. Other components do not depend on the internals of the execution engine and the data store. Thus, a new engine or data store does not require changes in other components such as the optimizer and the materializer.

Pipeline Implementation. Users can implement a new pipeline as a chain of operators, where operators have `transform`/`fit` methods. Currently, we support several existing Spark and scikit-learn preprocessing and model training operators. Some examples of the supported operators are imputation, min-max scaling, standard scaling, normalization, one-hot encoding, feature hashing, and polynomial feature generation. Implementation of user-defined operators requires the implementation of the `fit` and `transform` methods. The `fit`/`transform` APIs are common in data science libraries, allowing easy integration with existing tools. The only modification that users must make is to have the `fit` and model training operators return the computed statistics and trained models, which are then materialized in the data store.

4.3 Data Layout and API

The data store provides an abstraction around the underlying storage. In our current implementation, we utilize a common date-based partitioning of the data on HDFS. We also partition the materialized artifacts based on the same partitioning scheme. We store the materialized artifacts in an in-memory key-value store, where keys are

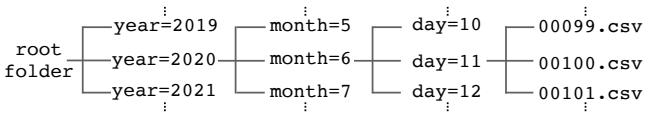


Figure 4: Example of the data layout on file system.

the partition names and values are the artifacts (i.e., Spark RDDs or Numpy arrays). Such partitioning strategies are common in data lake systems [47] as well as systems such as TFX [3]. Figure 4 shows an example of how the data store partitions CSV data on the file system. In Figure 4, we assume the smallest unit of time is a day.

The code in Listing 1 shows how to implement and schedule a pipeline. We start by defining the components of the pipeline (Lines 1-4). Then, we define the pipeline and set the execution engine to Spark (Lines 6-10). Finally, the pipeline manager schedules the pipeline for daily execution on the last 7 days of data (Line 12). Based on the current system time, the optimizer and the executor communicate with the data store and retrieve the appropriate data interval for execution.

```
1  data_parser = CSVParser()
2  feature_extractor = FeatureExtractor()
3  imputer = MeanImputer()
4  regression_model = LinearRegressionModel()
5
6  pipeline = Pipeline(id='simple-pipeline', engine='spark', components=[
7                                                    data_parser,
8                                                    feature_extractor,
9                                                    imputer,
10                                                   regression_model])
11 # schedule the pipeline to run every day on the last 7 day of data
12 manager.schedule(pipeline=pipeline, interval=Time.day(7), slide=Time.day(1))
```

Listing 1: Pipeline design and scheduling example.

5 MATERIALIZATION

Typical real-world ML applications contain many pipelines running on different data intervals. Since the input data size is typically large and the pipelines have many operators, the size of the generated artifacts becomes extremely large. In our experiments, when the size of the raw data is 24 GB, the total size of the artifacts can reach 2 TB. Thus, storing every artifact is not cost-efficient. In this section, we first define our cost model and materialization problem. Then, we describe our materialization algorithm. Table 1 shows a summary of the notations used in this section.

5.1 Problem Formulation

Cost Model. We define a scheduled task as $T : \langle p, s, i \rangle$, where p is a pipeline DAG and s is the slide, which indicates how frequently we execute the pipeline. For example, a pipeline that runs every day has a slide value of $s = 1$. Lastly, i represents the interval of input data in the form $i = [begin, end)$. For example, the three tasks for our example use case in Section 1 are $T_1 : \langle p_1, 1, [0, 30) \rangle$, $T_2 : \langle p_2, 1, [0, 10) \rangle$, and $T_3 : \langle p_3, 10, [0, 10) \rangle$. When s is smaller than the size of i, consecutive executions of a pipeline operate on overlapping data intervals resulting in redundant computation.

We denote a pipeline DAG p as the sequence of its operators (in topological order), $p = \langle o_1, o_2, ..., o_n \rangle$, where o_j $(1 \leq j \leq n)$ is an operator. When executing a pipeline, every operator o_j generates an intermediate artifact. We use o to also refer to the artifact generated by the operator. We denote the raw data by o_0 and the final ML model by o_n. An artifact can be divided into materialized and unmaterialized splits. For artifact o, m_o and u_o represent the intervals where o is materialized and unmaterialized, respectively.

Table 1: Table of notations.

Notation	Description
T, s, i	Scheduled task, slide, and interval
p	ML pipeline
o	Operator of a pipeline and its artifact
m_o, u_o	Materialized and unmaterialized splits of o
$C_{gen}(o, i)$	Cost of generating o in i
$C_{load}(o, m_o)$	Cost of loading the materialized split
$C_{exec}(o, u_o)$	Cost of executing the unmaterialized split
$C_{w\_exec}(o, i)$	Cost of training a warmstarted model
$C_{merge}(m_o, u_o)$	Cost of merging the splits

Because of the differences between the reuse procedures for model and non-model artifacts (described in Section 3), the cost of generating them is different. For simplicity, we use $C_{gen}(o, i)$ to represent the cost of generating both a model or a non-model artifact o in the interval i. Formula 1 presents the cost of generating a non-model artifact.

$$C_{gen}(o, i) = C_{load}(o, m_o) + C_{exec}(o, u_o) + C_{merge}(m_o, u_o) \quad (1)$$

$C_{load}(o, m_o)$ is the cost of loading the materialized split of o, if one exists. $C_{exec}(o, u_o)$ is the cost of executing o on the interval u_o, i.e., where o is unmaterialized. $C_{merge}(m_o, u_o)$ is the cost of merging the materialized and unmaterialized artifact splits of o. Note that for non-model artifacts, $m_o \cup u_o = i$ and $m_o \cap u_o = \emptyset$. Therefore, if a sub-interval of o is materialized, we do not need to recompute it.

Formula 2 shows the cost of generating a model artifact.

$$C_{gen}(o, i) = \begin{cases} C_{load}(o, m_o) + C_{w\_exec}(o, i) & \text{, if } m_o \text{ exists} \\ C_{exec}(o, i) & \text{, otherwise} \end{cases} \quad (2)$$

With or without reuse (i.e., model warmstarting), the model training process runs on the entire interval. When there is a materialized model artifact from an earlier execution, we load the model and execute the operator with a warmstarted materialized model. $C_{w\_exec}(o, i)$ represents the execution with a warmstarted model. If no materialized model exists, we execute the operator (train the model with random initialization of the parameters), which is denoted by $C_{exec}(o, i)$. Note that contrary to the non-model artifacts, the materialized interval m_o is not necessarily a sub-interval of i. This is because we can warmstart a model as long as the size and shape of the models (number of weight parameters) are equal.

The total cost of executing a pipeline p of n operators on interval i is the sum of the cost of generating every artifact of the pipeline.

$$C(p, i) = \sum_{o_j \in p} C_{gen}(o_j, i) \quad (3)$$

Problem Statement. We define the problem of artifact materialization as finding the optimal set of artifact splits to materialize in order to minimize the sum of the execution cost of all the scheduled tasks under a limited storage budget. More formally:

Definition 1. *Assume $\mathcal{T} = \{T_1, T_2, ..., T_{|\mathcal{T}|}\}$ is the set of scheduled tasks, where $T_t : \langle p_t, s_t, i_t \rangle$, $|p_t|$ is the number of artifacts in p_t, and β is the storage budget. We also define $\mathcal{M}$ as the set of all possible generated artifact splits of all the tasks. Then, the optimization problem is:*

$$\begin{aligned} &\underset{\mathcal{M}^* \subseteq \mathcal{M}}{\text{minimize}} \sum_{t=1}^{|\mathcal{T}|} \frac{C(p_t, i_t)}{s_t} \\ &\textit{subject to} \sum_{m \in \mathcal{M}^*} size(m) \leq \beta. \end{aligned} \tag{4}$$

The cost of a scheduled task is inversely proportional to its slide (the larger the slide the less frequently a pipeline is executed). Therefore, in Formula 4, we divide the execution cost of a pipeline by slide. For example, consider two tasks that have the same pipeline and interval, but different slides of 1 and 10, respectively. For every 10 executions of the first task, there is only one execution of the second task. Therefore, the overall cost of the second task is 10 times smaller. By dividing the cost by the slide, we can account for the impact of the slide in our optimization problem.

Cost Estimation. We profile the scheduled pipelines on a sample of the raw data to estimate the costs. Furthermore, we improve the estimates after every execution. Note that accurate cost estimation of complex ML operators is not trivial. However, our algorithms only require the estimated costs of the different operators to have the same order as the real costs. I.e., if the real cost of an operator is smaller than another operator, then its estimated cost should also be smaller. Given that we continuously update the estimated costs, we can be confident that the order of the estimated costs reflects the order of the real costs. In our experiments, we show that with our cost estimation approach, we can improve the performance of our system when compared to the state of the art.

5.2 Benefit-based Materialization

The goal of our materialization procedure is to find the optimal set of artifact splits to materialize, i.e., the solution to the materialization problem in Definition 1. Given the large space of operators, cost, size, and scheduling intervals, the problem of finding the optimal set of artifact intervals is complex. A simpler scenario where only batch data are considered is shown to be NP-hard [4]. We propose a heuristic-based algorithm, where the main idea is to materialize artifacts with a high cost-to-size ratio. Thus, ensuring the result of costly operators that require a smaller storage space is materialized.

Our proposed materialization procedure comprises two steps. In the first step, we combine the interval DAGs of all the scheduled pipelines into a global interval DAG, which we refer to as the *Materialization DAG*. The materialization DAG is an interval DAG, where—besides the operator and interval—every vertex, represented by v, contains the list of the pipeline identifiers they belong to, represented by $P(v)$. In the second step, we devise an algorithm that greedily materializes a vertex with the largest benefit score. The benefit score is computed in such a way that a higher score indicates a larger reduction in the execution cost if the artifact is materialized. Given the set of already materialized vertices, our greedy algorithm needs to recompute the scores. However, materializing a vertex does not affect the benefit score of every vertex in the materialization DAG. Therefore, to speed up the benefit recomputation, we propose an optimization that first detects the set of vertices that are impacted by the already materialized set and only recomputes the benefit of those vertices incrementally. The algorithm stops when the storage budget is exhausted. Algorithm 1 shows the overall process of our proposed materialization procedure.

In the rest of this section, we use the pipelines p_1 and p_2 from our running example of Section 1. We omit p_3 to simplify the description and the presentation of our materialization algorithm. For a scheduled task, the beginning and end of the interval are relative to the current system time, e.g., the interval $[0, 30)$ days shows the interval that started 30 days ago. The scheduled tasks of our running example are $T_1 : \langle p_1, 1, [0, 30) \rangle$ and $T_2 : \langle p_2, 1, [0, 10) \rangle$.

Algorithm 1: Artifact materialization algorithm.

Input: *IDs* Interval DAGs of the tasks, $\mathcal{B}$ storage budget
Output: $\mathcal{MV}$ set of materialized artifacts

1 $IDs \coloneqq sort(IDs)$; // sort by interval size
2 $G_{\mathcal{M}} \coloneqq \emptyset$; // initialize Materialization DAG
3 **for** $ID \leftarrow IDs$ **do**
4 update($G_{\mathcal{M}}, ID$)
5 $S \coloneqq 0$; // size of the materialized artifacts
6 **for** $v \leftarrow G_{\mathcal{M}}$ **do**
7 $benefits(v) \coloneqq$ benefit of v; // use Formula 5
8 **while** *true* **do**
9 $v \coloneqq argmax(benefits)$; // vertex with max *benefit*
10 $benefits.remove(v)$;
11 $S \coloneqq S + size(v)$;
12 **if** $S \geq \mathcal{B}$ **then**
13 break;
14 $\mathcal{MV} \coloneqq \mathcal{MV} \cup v$;
15 $recompute(benefits)$; // benefit recomputation
16 return $\mathcal{MV}$;

Materialization DAG Construction. The first step of the materialization procedure is to construct the materialization DAG. Algorithm 1 (Lines 1 - 4) shows the materialization DAG construction steps. The procedure sorts the tasks based on their interval size (increasing order). Then, starting with an empty materialization DAG, the procedure updates the materialization DAG using the sorted interval DAGs. Figure 5a shows the interval DAGs of p_1 and p_2, which we refer to as ID_1 and ID_2. For every vertex of an interval DAG (starting from the source vertex), one of the following four scenarios might occur while updating the materialization DAG.

(S1) The vertex does not exist in the materialization DAG. In this scenario, we add the vertex to the materialization DAG and add edges to the parents of the vertex. Every vertex in Figure 5b belongs to S1. (S2) The vertex partially matches an existing vertex. I.e., the operator of the vertex exists in the materialization DAG, however, the interval is larger than the interval of the vertex inside the materialization DAG. In this scenario, we split the vertex into two. The first vertex has the exact interval as the existing vertex in the materialization DAG. Instead of creating a new vertex, we update $P(v)$ of the existing vertex in the materialization DAG by adding the pipeline identifier of the new vertex. Then, we add the second vertex to the materialization DAG. Figure 5c demonstrates

an example of S2 (e.g., notice how ⟨FG-t, [0, 30)⟩ of ID_1 is split into ⟨FG-t, [0, 10)⟩ and ⟨FG-t, [10, 30)⟩ in the materialization DAG). (S3) The vertex does not exist in the materialization DAG, however, its parent was part of S2. In this scenario, we also split the vertex into two, in such a way that the resulting intervals match the intervals of the vertices resulted from S2. Figure 5c shows how ⟨Norm-t, [0, 30)⟩ of ID_1 is split into ⟨Norm-t, [0, 10)⟩ and ⟨Norm-t, [10, 30)⟩. (S4) The vertex is a last-level ML model. In this scenario, first, the vertex representing the model is added to the materialization DAG. Then, if the parent of the ML model was split earlier (because of S2 or S3), edges are drawn from all the vertices to the ML model.

Benefit Computation. In the second step of the materialization procedure, we utilize a greedy algorithm that iteratively computes a benefit score and materializes the vertex with the highest score (Lines 6 and 7 of Algorithm 1). Let $v = \langle o, i \rangle$ be a vertex in the materialization DAG. We define the materialization benefit of v as the reduction in the cost of executing the pipelines when v is materialized. Formula 5 computes the materialization benefit of v.

$$\text{benefit}(v) = \frac{C(v)}{S(v)} * \sum_{p \in P(v)} L(p) \tag{5}$$

In Formula 5, $C(v)$ and $S(v)$ represent the estimated compute and storage costs of v. $C(v)$ is the sum of the execution cost of all the operators from source to v in the materialization DAG. $S(v)$ represents the cost of storing the result of o in the interval i. The term $L(p)$ is the pipeline redundancy ratio. We define $L(p)$ as the number of times p processes a single data point, $L(p) = \frac{|i|}{s}$, i.e., size of the interval divided by the slide. For example, for pipeline p_1, the redundancy ratio is $\frac{30}{1} = 30$, since every day of the data will appear in 30 consecutive executions. Intuitively, an artifact interval with a high benefit value indicates one or both of the following cases. First, the cost-to-size ratio of the artifact is larger than that of the other artifacts; therefore, materializing it results in a larger relative improvement in execution cost. Second, the interval of the artifact has a high data processing redundancy; therefore, materializing it eliminates the redundancy and improves the execution cost. Once the algorithm computes the benefit of all the vertices in the materialization DAG, it materializes the vertex with the highest benefit score.

Benefit Recomputation. After materializing a vertex, the algorithm must recompute the benefit of the other vertices. We show that materializing a vertex *only* affects the benefits of its ancestors and descendants, and the algorithm only needs to adjust the benefits of such vertices.

Materialized vertices are directly loaded from the artifact store. Therefore, a materialized vertex may break the dependency of its descendants on the ancestors of the materialized vertex, which reduces the materialization benefit of the ancestors. Let v_m be the materialized vertex. If removing v_m from the materialization DAG disconnects the descendants of v_m from its ancestors, then we remove every pipeline p from $P(v_a)$, where $p \in P(v_m)$ and v_a is an ancestor of v_m. This is because the ML model in p where $p \in P(v_m)$ no longer requires the ancestors of v_m. This impacts the term $\sum_{p \in P(v)} L(p)$ in Formula 5.

For example, assume the algorithm has materialized the vertex $v_m = \langle$Var-t, $[0, 10)\rangle$ in Figure 5c. Since $P(v_m) = \{p_2\}$, the

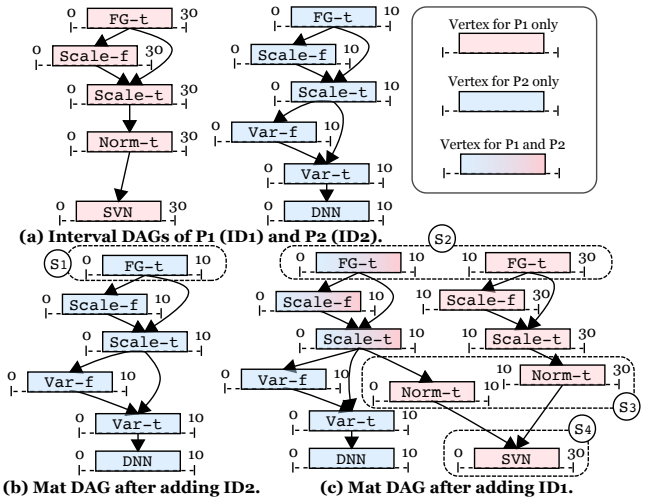


Figure 5: The process of materialization DAG construction.

algorithm removes p_2 from all of its ancestors, i.e., ⟨Var-f, [0, 10)⟩, ⟨Scale-t, [0, 10)⟩, ⟨Scale-f, [0, 10)⟩, and ⟨FG-t, [0, 10)⟩. Note that materializing a vertex such as ⟨Var-f, [0, 10)⟩ does not break the dependency of its descendants (e.g., ⟨Var-t, [0, 10)⟩) from its ancestors (e.g., ⟨Scale-t, [0, 10)⟩), since there are more than one path connecting the ancestors to the descendants. Therefore, it has no impact on the benefit of its ancestors.

For descendants of a materialized vertex, we need to adjust the compute cost, i.e., $C(v)$. $C(v)$ is the sum of the execution cost of the vertices from source to v. If a materialized vertex breaks the dependency of its ancestors on its descendants, then the materialized vertex becomes a pseudo-source for its descendants. Thus, the algorithm recomputes the compute cost of all the descendants starting from the materialized vertex. Lines 8 - 15 of Algorithm 1 show the process of benefit recomputation and vertex materialization. For example, materializing ⟨Scale-t, [10, 30)⟩ in Figure 5c breaks the dependency of the vertices ⟨Norm-t, [10, 30)⟩ and ⟨SVM, [0, 30)⟩ on ⟨FG-t, [10, 30)⟩ and ⟨Scale-f, [10, 30)⟩. Furthermore, if a materialized vertex does not break the dependency, it still impacts the total compute costs of its descendants. The execution cost of a materialized vertex is zero, which reduces the total compute cost of its descendants. For example, if ⟨Var-f, [0, 10)⟩ is materialized, we set its compute cost to zero before computing the cost of ⟨Var-t, [0, 10)⟩ and ⟨DNN, [0, 10)⟩. After the recomputation, the node with the highest benefit value is materialized. The greedy algorithm stops when the storage budget is exhausted.

6 REUSE AND PLAN GENERATION

We now introduce our reuse and execution plan generation algorithm (for brevity, we refer to the algorithm as the *reuse algorithm*). We define the reuse problem as follows. Given a set of materialized artifacts (the output of the materialization algorithm of Section 5) and the set of triggered pipelines, i.e., the pipelines that will be executed, find the optimal execution plan. The execution plan dictates what intervals of the materialized artifacts to reuse and what intervals the pipeline operators should be executed on.

The simplest approach to execute the pipelines is to run them one by one and reuse materialized artifacts from the artifact store. However, this results in redundant data processing and data loading, as many pipelines share similar operators. Our reuse algorithm follows three principles. First, it reuses materialized artifacts when possible. Second, inspired by multi-query optimization [9], our algorithm shares raw data, materialized artifacts, and the output of operators to eliminate redundant computation. Third, the algorithm ensures correctness in the presence of a distribution shift. In this section, we first describe our reuse algorithm for generating an execution plan. Then, we describe the process of model reuse. Lastly, we explain how we handle distribution shifts in data.

Algorithm 2: Artifact reuse algorithm.

Input: IDs Interval DAGs of the tasks
Output: G_E Execution DAG

1 $IDs \coloneqq sort(IDs, decreasing = True)$; // reverse sort
2 $G_E \coloneqq \emptyset$; // initialize Execution DAG
3 **for** $ID \leftarrow IDs$ **do**
4 update(G_E, ID); // update execution DAG and split nodes if Inequality 6 holds
5 **for** $v \leftarrow backward\_traversal(G_E)$ **do**
6 **if** v *is not model* **then**
7 **if** v *is materialized* **then**
8 remove incoming edges of v ; // edge pruning
9 $v_i = \bigcup_{u \in successors(G_E, v)} u_i$; // interval shrinking
10 return G_E

6.1 Reuse Algorithm

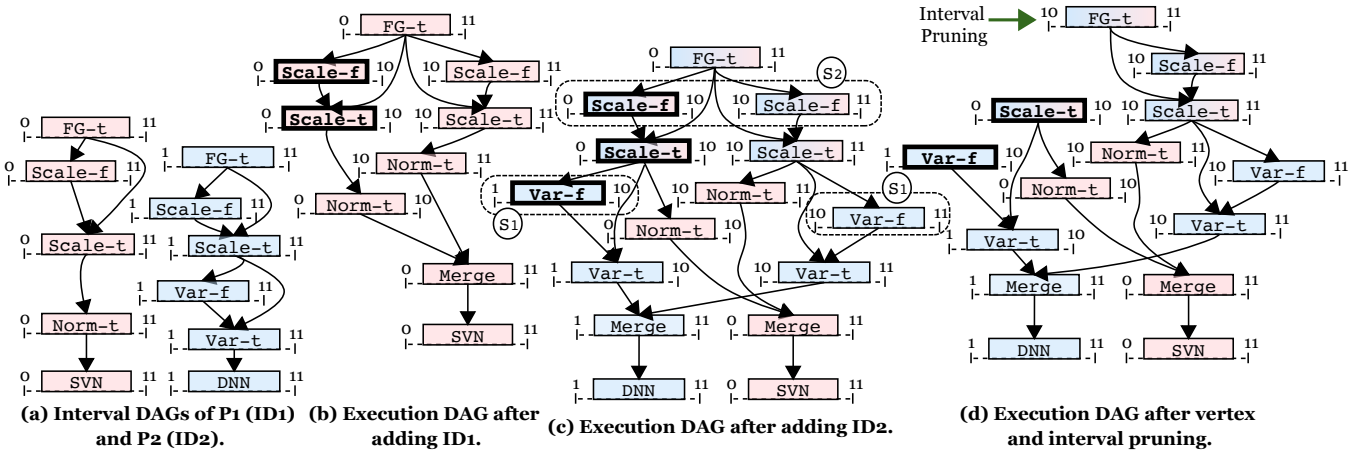
The input to our reuse algorithm is a set of triggered pipelines and the output is an execution plan in DAG form, which we refer to as the *execution DAG*. The reuse algorithm has three main steps. First, the algorithm constructs the interval DAG for every pipeline. The vertices of the interval DAG have the operators and the data interval that will be used during the execution. All the vertices of an interval DAG must have the same interval. For example, if a pipeline will be executed on the interval $[1, 11)$, all of the vertices in the interval DAG have the interval $[1, 11)$. Second, the algorithm sorts the interval DAGs by the size of their intervals in *decreasing order* and adds them one by one to the execution interval DAG. During this step, the algorithm consults the data store for available materialized artifacts and split the vertices into materialized and unmaterialized sub-intervals. Third, the algorithm traverses the execution DAG backward to prune the incoming edges of the materialized vertices, since they are loaded from the artifact store and are no longer computed. Furthermore, some sub-intervals of the vertices that have materialized descendants may not be necessary anymore and the algorithm shrinks their intervals. Algorithm 2 shows the overall process of our reuse and execution plan generation algorithm. Figure 6 illustrates the main steps of our reuse algorithm, where the triggered pipelines are p_1 and p_2 of our running example. In the figure, we assume that eleven days have passed since the start of the system. Therefore, for p_2, the interval for training is $[1, 11)$ (the last 10 days) and for p_1, the interval is $[0, 11)$, since the scheduled interval of p_1 (30 days) is larger than the available data (Figure 6a). In this example, we assume that the intervals `Scale-f` : $[0, 10)$, `Scale-t` : $[0, 10)$, and `Var-f` : $[0, 10)$ have been materialized.

Execution DAG Construction. The reuse algorithm sorts the interval DAGs based on the interval sizes (by decreasing order), initializes the execution DAG, and updates the DAG by adding the interval DAGs one by one (Lines 1 to 4 of Algorithm 2). For every vertex of the interval DAG, if a sub-interval of it is materialized, then the algorithm splits the vertex into multiple vertices. Each new vertex represents a continuous sub-interval where the data is either materialized or unmaterialized. Figure 6b shows the execution DAG after adding ID_1. Since `Scale-f` : $[0, 10)$ is materialized, the vertex $\langle$`Scale-f`, $[0, 11)\rangle$ is split into $\langle$`Scale-f`, $[0, 10)\rangle$ and $\langle$`Scale-f`, $[10, 11)\rangle$, where $\langle$`Scale-f`, $[0, 10)\rangle$ is materialized (as shown by bold border in the figure). Similarly, $\langle$`Scale-t`, $[0, 11)\rangle$ is also split into two vertices. More formally, the algorithm splits a vertex $v = \langle o, i\rangle$ into a set of vertices, represented by $SPLITS(\langle o, i\rangle)$. Every vertex in $SPLITS(\langle o, i\rangle)$ is a tuple of three elements $\langle o, i', ms\rangle$, where o is the same operator as the original vertex, i' is a sub-interval of i, and ms is an indicator variable, which is one when the artifact in the sub-interval i' is materialized and zero otherwise. The resulting sub-intervals of the vertices in $SPLITS(\langle o, i\rangle)$ do not overlap and their union is equal to i.

It is important to note that the goal of the algorithm is to minimize the cost function in Formula 3 of Section 5.1. Splitting an interval into materialized and unmaterialized sub-intervals reduces the execution cost. However, according to our cost functions, loading the materialized interval and merging the intervals incur extra costs. Similar to our cost function, let C_{exec} be the cost of executing an operator in an interval, C_{load} be the cost of loading an artifact in a given interval, and C_{merge} be the cost of merging all the materialized and unmaterialized sub-intervals. The reuse algorithm only splits the intervals of a vertex when Inequality 6 holds, which indicates that the total cost when using a materialized sub-interval, i.e., cost of loading the materialized sub-intervals, executing the operator in the unmaterialized sub-interval, and merging all the sub-intervals, is smaller than the cost of executing the operator on the entire interval.

$$C_{exec}(o, i) > \sum_{\langle o, i', ms\rangle \in SPLITS(\langle o, i\rangle)} \Big[C_{load}(o, i') * ms + C_{exec}(o, i') * (1 - ms) \Big] + C_{merge}(SPLITS(\langle o, i\rangle)) \quad (6)$$

The algorithm continues adding every interval DAG. If the execution DAG is not empty, for every new vertex, one of the two scenarios occurs. (S1) No matching vertex exists in the execution DAG. In this scenario, the algorithm proceeds as earlier, i.e., adding the vertex to the execution interval DAG and splitting it if there are materialized sub-intervals and Inequality 6 holds. For example, the vertex $\langle$`Var-f`, $[1, 11)\rangle$ of ID_2 is split into $\langle$`Var-f`, $[1, 10)\rangle$ and $\langle$`Var-f`, $[10, 11)\rangle$ in Figure 6c. (S2) There exists a matching vertex with full or partial interval overlap. In this scenario, the algorithm does not add new vertices since there are existing vertices that match the vertex. Since the algorithm adds vertices sorted by their interval in decreasing order, any interval of the current interval DAG is smaller or equal to the interval in the execution DAG. Therefore, the algorithm does not need to add or update the vertices. For



(a) Interval DAGs of P1 (ID1) and P2 (ID2). (b) Execution DAG after adding ID1. (c) Execution DAG after adding ID2. (d) Execution DAG after vertex and interval pruning.

Figure 6: The process of the reuse algorithm. Vertices with bold border and text indicate materialized vertices.

example, the result of splitting ⟨Scale-f, [1, 11)⟩ of ID_2 already exist in the execution DAG (⟨Scale-f, [0, 10)⟩ and ⟨Scale-f, [10, 11)⟩ in Figure 6c). Before training a model, the algorithm merges the sub-intervals of the last non-model operators.

Execution DAG Pruning. After adding all the interval DAGs of the triggered pipelines, the reuse algorithm traverses the execution DAG backward starting from the last non-model artifact and performs two types of pruning. First, the algorithm prunes unnecessary edges and vertices from the execution DAG. Materialized artifacts are loaded from the artifact store and are independent of their predecessors. Thus, the reuse algorithm prunes the incoming edges of the materialized artifact. Vertices that have no incoming or outgoing edges are removed from the execution DAG. Such a scenario happens when a materialized artifact, i.e., no incoming edges, only has materialized successors, i.e., no outgoing edges. The second pruning operation shrinks the sub-interval of the vertices that are not required. During the backward traversal, the reuse algorithm propagates the interval of every unmaterialized vertex to its predecessors. For every vertex visit during the traversal, the reuse algorithm sets the interval of the vertex to the union of all the intervals propagated from its successors. Using this approach, only the interval of the data that is required by the downstream vertices will be computed. Algorithm 2 (Lines 5-9) shows the backward traversal, pruning, and shrinking processes of our reuse algorithm.

Figure 6 (c and d) shows the execution DAG before and after pruning. Since ⟨Var-f, [1, 10)⟩ and ⟨Scale-t, [0, 10)⟩ are materialized, we prune their incoming edges. We also completely remove vertex ⟨Scale-f, [0, 10)⟩, since the vertex itself and its successor are materialized. Because of pruning some of the edges and vertices, only the interval [10, 11) is propagated to vertex ⟨FG-t, [0, 11)⟩. Therefore, the algorithm prunes sub-interval [0, 10) from the vertices resulting in the final pruned execution DAG.

Reusing Model Artifacts. We support model reuse through warmstarting the model training. When there are materialized models, the reuse algorithm initializes the training procedure with the weight parameters of the materialized model. If multiple models in the artifact store overlap with the model in the execution DAG, the algorithm selects the latest model for warmstarting.

Complexity Analysis. In some applications, there are hundreds of parallel pipelines. Our reuse algorithm runs in linear time and only makes two passes over the interval DAG of every scheduled task. In our experiments, we show that the reuse algorithm incurs negligible overhead and can scale to hundreds of parallel pipelines.

6.2 Handling Distribution Shift

When a distribution shift occurs, reusing materialized artifacts may generate inconsistent results. This is particularly the case for operators that generate feature artifacts and receive statistics artifacts as input. Consider the case when scaled features resulting from standard scaling are materialized. The presence of a distribution shift indicates that the old materialized features were scaled with a different mean and variance. Thus, reusing such materialized features generates inconsistent output that hurts the final accuracy of the ML models. Therefore, in such a scenario, the reuse algorithm only splits vertices until (and including) the *first* statistics-based operator (e.g., `fit` or other aggregation operators) of every path starting from the source vertex. This is because a distribution shift only impacts the statistics-based and model training components. Generated feature artifacts of the operators that are not statistics-based are agnostic to the change in the distribution. As a result, we can still reuse such materialized artifacts in consecutive executions.

In Figure 6, the operator FG-t (polynomial feature generator) is not statistics-based, which makes it agnostic to a distribution shift. Similarly, the operator Scale-f, which computes some statistics over generated features of FG-t is not impacted by the distribution shift (since its input is agnostic to the distribution shift). Therefore, in presence of a distribution shift, the reuse algorithm is only allowed to reuse the materialized vertex ⟨Scale-f, [0, 10)⟩.

7 EVALUATION

In this section, we experimentally evaluate: (i) the end-to-end performance of our system, (ii) the effect of distribution shift on model accuracy, and (iii) the overhead of our optimizations. Our results show that our solution achieves up to an order of magnitude better performance compared to current practices while maintaining the

Table 2: Details of datasets, pipelines, use case schedules, and dataset and artifact sizes. Ensemble and Interval Tuning are two use cases that we evaluate in the experiments. The respective columns show the task schedules of each use case.

Dataset	Pipelines	Ensemble	Interval Tuning	Data Size
Higgs [2]	A total of 6 pipelines. Three with imputaion→scaling→SVM with varying regularization $\alpha = \{0.01, 0.001, 0.0001\}$. Three with imputation→polynomial feature generation→scaling→SVM with varying $\alpha = \{0.01, 0.001, 0.0001\}$	Daily on the last 7 days	Daily on the last 3, 5, and 7 days	Raw = 4 GB Total Artifact = 100 GB
URL [34]	A total of 6 pipelines. The pipelines contain imputation→scaling→feature, hashing→SVM model. There are three different hash sizes (10k, 25k, and 50k) and 2 different regularization values $\alpha = \{0.01, 0.001\}$ resulting in 6 total pipelines	Daily on the last 7 days	Daily on the last 7, 14, and 21 days	Raw = 2.2 GB Total Artifact = 4 GB
Taxi [50]	A total of 10 pipelines based on the top performing notebooks of the NYC Taxi Duration kaggle contest [5, 37, 39]. The pipelines have different operators such as date and distance extraction, anomaly detection, one-hot encoding, scaling, normalizing, and linear regression with different regularization values.	Monthly on the last 6 months	- - - - - -	Raw = 24 GB Total Artifact = 2 TB

model accuracy when data distribution changes. In addition, our optimizations incur negligible overhead (less than 0.01%).

7.1 Setup

Prototype. We implement a prototype of our system in Python using two different execution engines: a single-node (Python 3.7) and a parallel execution (Spark 3.1.2 [58]) engine. For our Python engine, we extend several sklearn models and preprocessing components [41]. For our Spark engine, we utilize its ML library [38] to implement models and preprocessing components.

Our reuse algorithm relies on a `merge` method to combine the sub-intervals of feature and statistics artifacts. In the Python engine, we represent feature artifacts as NumPy arrays [23] and use the built-in `concatenate` for merging them. In the Spark engine, we represent feature artifacts as RDDs [57] and use the built-in `union` function of Spark for merging them. The implementation of the `merge` method for the statistics artifacts depends on the type of statistics. Currently, we implement the `merge` method for several statistics, such as count, mean, and variance. For user-defined operators, the user must implement the `merge` method. We utilize the `persist` method of RDDs to materialize the artifacts in memory.

For our single-node (Python) experiments, we use a Linux Ubuntu machine with 128 GB of RAM. For our Spark experiments, we utilize an 11-node (one master and 10 workers) cluster, with each node having an Intel Xeon 2.4 GHz 16 cores and 28 GB of dedicated RAM.

Systems and Baselines. We evaluate the performance of different deployment scenarios on our system and other baselines. *MaR* is our system, which includes our novel DAG-based materialization and reuse algorithms. *TFX* is an implementation of a deployment system based on TFX [3]. TFX executes the pipelines one by one and, for every pipeline, it materializes the results of the last operator. *Baseline* represents the most common current practice where pipelines are executed one by one without any materialization.

Datasets and Pipelines. We use three datasets and design several ML pipelines for each one. For two of the datasets, we consider two common multi-pipeline use cases, i.e., an ensemble of pipelines and interval tuning. For the third dataset, we only show the results for the ensemble use case as the results for the interval tuning follow the same patterns. Table 2 shows the datasets, pipelines, and their schedules for each use case.

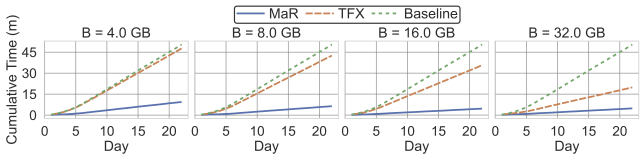
Higgs [2] is a static dataset for a classification task. We manually divide the dataset into 22 equal chunks, where each chunk represents one day and contains 500,000 data points. The total size of the raw data is around 4 GB. We design 6 different pipelines to process the Higgs dataset. In the ensemble use case, we run all the pipelines every day on the last week (7 days) of the data. In the interval tuning use case, we select the two top-performing pipelines and execute them every day on the last 3, 5, and 7 days of data, respectively (resulting in a total of 6 different tasks). Depending on the ML pipelines, the total size of the intermediate artifacts can reach up to 100 GB. We contribute the large size of the intermediate artifacts to the polynomial feature generation operator.

The *URL* dataset [34] is a collection of URLs—labeled as malicious or legitimate—and their lexical and host-based features gathered for 121 days. The dataset contains both numerical and categorical entries. We design 6 pipelines for processing the URL dataset. We utilize the feature hasher of sklearn to generate sparse features out of the categorical entries. The URL dataset contains a large number of features (between 10,000 to 50,000). Since we store features as sparse vectors, the actual size of the generated artifacts is small (up to 4 GB). In the interval tuning use case, we select the two top-performing pipelines and execute them every day on the last 7, 14, and 21 days. The URL dataset has distribution shifts, and we primarily use it to investigate the performance of our system under distribution changes. We use the single-node engine (Python) for running all the experiments for the Higgs and URL datasets.

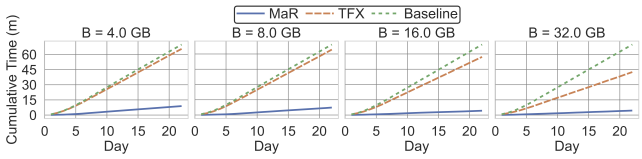
The *Taxi* dataset [50] contains records of Taxi rides in New York. We use the data gathered in the year 2015 to evaluate our parallel execution engine (Spark). Based on a Kaggle competition [28], we utilize this dataset for estimating the trip duration of every taxi ride (a regression task). We design 10 pipelines based on the publicly available top-performing solutions of the Kaggle competition. The size of the raw data is roughly 24 GB and the total size of the intermediate artifacts of all the pipelines is approximately 2 TB.

7.2 End-to-end Performance

In this section, we analyze the performance under varying budgets and the number of parallel tasks. Here, we disable warmstarting and use the same random seed for the model training to ensure the training time remains the same, in order to focus on the impact of

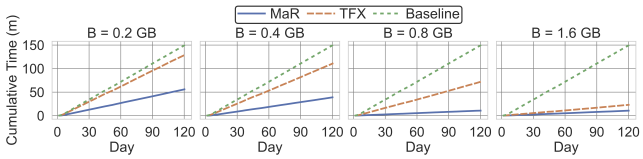


(a) Ensemble use case with different budgets.

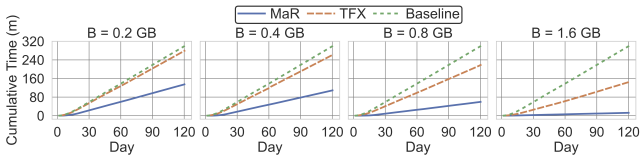


(b) Interval tuning use case with different budgets.

Figure 7: Cumulative run time for the Higgs dataset.



(a) Ensemble use case with different budgets.



(b) Interval tuning use case with different budgets.

Figure 8: Cumulative run time for the URL dataset.

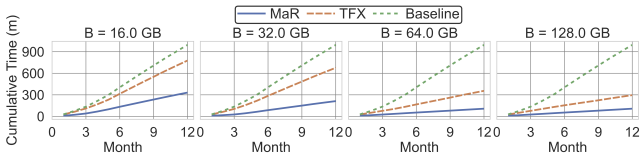


Figure 9: Cumulative run time for the Taxi dataset.

materializing and reusing statistics and feature artifacts. Therefore, we do not include the model training time in the reported figures.

Cumulative Cost With Different Budgets. Here, we investigate the end-to-end performance using varying budgets.

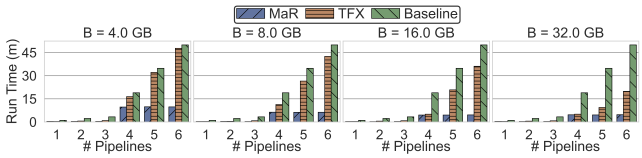
Figure 7 shows the cumulative run time of both use cases for the Higgs dataset. Since the pipelines in the ensemble use case are highly similar (Figure 7a), even with a small budget, MaR outperforms Baseline and TFX by a factor of 5. MaR identifies the two unique prefixes, i.e., impute→scale for the first three pipelines and impute→poly feature gen→scale for the second three pipelines (Table 2), and materializes their results only once. However, TFX materializes the artifacts of each pipeline separately, which quickly exhausts the budget. Therefore, with a small budget, TFX only materializes a small fraction of the artifacts and incurs a run time similar to Baseline. As the budget increases, TFX materializes a larger fraction of artifacts. When the budget is 32 GB, TFX materializes last-level features of all but one pipeline (it requires 38 GB to materialize all the last-level features). Therefore, it outperforms Baseline. However, MaR still outperforms TFX by a factor of 4 thanks to its DAG-based execution, which shares operators' execution and loading of the materialized artifacts. Baseline does not depend on the materialization budget and incurs the same run time.

In the interval tuning use case (Figure 7b), MaR outperforms both TFX and Baseline with a larger margin. With a budget of 4 GB, MaR outperforms Baseline and TFX by a factor of 8. For interval tuning, only the two top-performing pipelines with different intervals are used. The polynomial feature generator component, which is in the two top-performing pipelines, generates a large portion of the artifacts. Therefore, the total size of the generated artifacts for the interval tuning use case (approximately 100 GB) is larger than the ensemble use case (approximately 80 GB). TFX requires 52 GB to materialize the last-level features of the pipelines in the interval tuning use case. As a result, with a budget of 16 GB and 32 GB, the run time of MaR is 13 and 10 times smaller than TFX. MaR reaches its peak performance with a budget of 16 GB, i.e., MaR materializes all the artifacts that can be reused. However, with a budget of 16 GB, TFX only materializes a fraction of the artifacts.

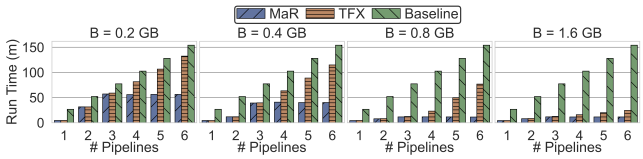
Figure 8 shows the cumulative run times for the URL dataset. The budget varies from 0.2 to 1.6 GB, i.e., 5% to 40% of the total generated artifacts as it was for the Higgs dataset. In the ensemble use case (Figure 8a), with a budget of 0.2 GB, MaR outperforms TFX and Baseline by factors of 2.5 and 3. For this use case, TFX materializes all the last-level features with a budget of 1.6 GB. However, MaR still outperforms TFX by a factor of 2. Although TFX materializes everything, it still executes each pipeline separately. However, MaR utilizes a DAG-based execution where similar operators of different pipelines are executed once.

Figure 8b shows that for the interval tuning use case, MaR outperforms TFX and Baseline by a larger margin when compared to the ensemble use case. There are two reasons for this better relative performance. First, since intervals are larger (i.e., 7, 14, and 21 days), TFX requires 2.6 GB to materialize all the last-level feature artifacts (compared to 1.6 GB for the ensemble use case). Second, our materialization algorithm quickly materializes the sub-interval [0, 7), i.e., the last 7 days of the data, which is shared between all the tasks, and materializing it greatly reduces the run time. TFX materializes the last-level feature artifacts of each task separately, which generates redundant data. When the budget is 1.6 GB, MaR nearly materializes all reusable artifacts. However, TFX only materializes the artifacts of the first four pipelines and has to completely execute the two last ones on their entire intervals. As a result, MaR outperforms TFX by one order of magnitude with a budget of 1.6 GB.

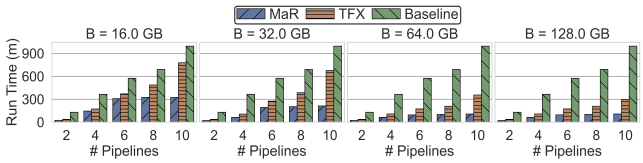
Figure 9 shows the cumulative run times of the ensemble use case for the Taxi dataset. Similar to the other datasets, MaR outperforms Baseline and TFX by a factor of 2.5 and 3 when the budget is small (16 GB) and by a factor of 3 and 9 when the budget is very large (128 GB). Two reasons contribute to the better performance of MaR when compared to TFX. First, TFX requires a very large budget to materialize all the last-level features, i.e., 80 GB. Whereas, with a budget of approximately 40 GB, MaR can identify all the common



(a) Higgs ensemble use case with different budgets.



(b) URL ensemble use case with different budgets.



(c) Taxi ensemble use case with different budgets.

Figure 10: Total run time with different number of parallel pipelines for the ensemble use case.

pipeline prefixes and materialize them. Second, the DAG-based execution of MaR eliminates any redundant execution. As a result, even with a large budget of 128 GB—where TFX can materialize all the last-level features—MaR still outperforms TFX by a factor of 3.

Number of Parallel Tasks. We now evaluate the performance of our system when increasing the number of parallel pipelines.

For the Higgs ensemble use case (Figure 10a), the first three pipelines and the second three pipelines have similar data processing steps (Table 2). Therefore, MaR has the same run time when the number of pipelines is 1 to 3 or 4 to 6, respectively. This is because MaR runs the DAG of the pipelines and shares the execution of similar operators. TFX, however, runs each pipeline separately. Therefore, with a budget of 4.0 GB, TFX runs all 6 tasks in 48 min (6x larger than MaR). When the budget is 32 GB, TFX nearly materializes all of the last-level features. But, it still runs each task separately. Therefore, MaR outperforms TFX by a factor of 4.

For the URL ensemble use case (Figure 10b), the data processing components of the first three pipelines are the same as the data processing components of the second three pipelines (Table 2). Therefore, MaR has the same run time when executing 3, 4, 5, or 6 pipelines, regardless of the materialization budget. However, as TFX executes the pipelines separately, even with a budget of 1.6 GB where all the last-level features are materialized, it incurs a total run time twice higher than MaR. MaR has the best relative improvement compared to TFX when the budget is 0.8 GB. This is because, at this budget, MaR can materialize all the reusable artifacts while TFX materializes the generated features of the first 4 pipelines. As a result, when the budget is 0.8 GB and the number of parallel pipelines is 6, MaR outperforms TFX by a factor of 6.

We observe a similar pattern for the Taxi ensemble use case (Figure 10c). Thanks to the DAG-based execution, MaR outperforms

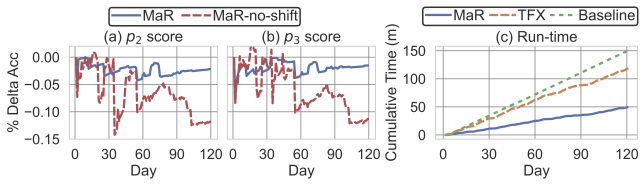


Figure 11: Distribution shift effect on accuracy and run time of the URL ensemble use case.

TFX and baseline by a factor of 2.5 and 3 for small budgets (e.g., 16 GB), and by a factor 3 and 9 for large budgets (e.g., 128 GB).

For the interval tuning use case for the URL and Higgs datasets, we observe a similar trend. Since the size of the artifacts in the interval tuning use case is larger, the relative performance improvement of MaR is even greater than in the ensemble use case, i.e., for the maximum budget and 6 parallel pipelines, MaR outperforms TFX by an order-of-magnitude for both URL and Higgs interval tuning use cases. We omit the graphs due to space limitations.

7.3 Impact of Distribution Shift

In this experiment, we enable distribution shift detection for MaR and examine the run time and quality of the models for URL data.

Accuracy. To show the impact of the distribution shift, in Figures 11a and 11b, we plot the difference between the accuracy of MaR, with and without shift detection, to Baseline. Since Baseline does not reuse any materialized data, we assume distribution shift does not impact its performance. For the URL ensemble use case, we show the performance of the best pipelines (i.e., p_2 and p_3). We use the prequential technique [10] to report the accuracy of the models, i.e., we compute accuracy on the next day of data. With shift detection, the accuracy of the models is close to Baseline (average of 0.02% difference). Note that the model performance depends on the shift detection technique. A technique that guarantees to find a distribution shift improves the accuracy. MaR with no shift detection reduces the model accuracy by 0.15%. This is because as data distribution changes, reusing old artifacts generates inconsistent results, which negatively impacts the model performance.

Run time. Assuming that one implements the same detection mechanism in TFX so that the accuracy does not drop, we compare the run time of our system with TFX. Note that the original TFX system includes a mechanism for detecting distribution shifts. However, it requires the user to manually invalidate the materialized artifacts. Figure 11c shows the run time (data processing components) of MaR and TFX, with shift detection, for the URL ensemble use case. Out of 121 days, 70 days contain distribution shifts. As a result, for 70 days no reuse was performed. MaR outperforms TFX and baseline by a factor of 2.3 and 3, respectively. This is mainly contributed to the DAG-based execution of the pipelines since in nearly more than half of the days no reuse was performed.

7.4 Impact of Model Reuse

We facilitate model reuse via warmstarting the model training. The main advantages of MaR for warmstarting are: (1) MaR offers warmstarting out-of-the-box, and in multi-pipeline scenarios, it

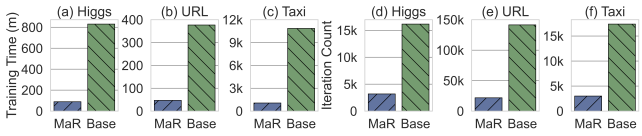


Figure 12: Effect of warmstarting on training time and iteration counts (Base represents Baseline).

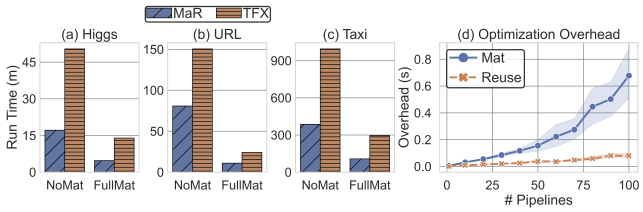


Figure 13: Total run time with no materialization and full materialization. (e) Overhead of materialization and reuse.

automatically selects the matching model for warmstarting. (2) In the presence of a distribution shift, MaR invalidates the materialized models and chooses not to warmstart the models. TFX also offers warmstarting. However, it requires the user to select a model from the set of stored models. Therefore, the impact of warmstarting for both TFX and MaR is similar and we only show the results for MaR.

In Figure 12, we show the impact of warmstarting on the total training time (we exclude the data processing time) and the total number of training iterations for Higgs, URL, and Taxi ensemble use cases. The values show the total sum of training time and iteration counts for all pipelines for the entire deployment period. Since model artifacts are typically smaller than the feature artifacts and require large compute costs, our materialization algorithm assigns a large benefit score to the models and always materializes them. In the figure, we only show the total training time for budgets of 4 GB, 0.2 GB, and 16 GB for Higgs, URL, and Taxi, respectively. This is because larger materialization budgets have no impact on the training time. Model warmstarting using a previously converged model greatly reduces the number of iterations until convergence, as shown in Figures 12d, 12e, and 12f. As a result, MaR outperforms Baseline by up to an order-of-magnitude for all three use cases.

Note that the run time of model training can be higher than data processing (e.g., deep learning models [18]). Thus, the benefit of model reuse (i.e., warmstarting) may overshadow the benefit of non-model artifact reuse. However, many common industrial scenarios involve simple or incremental models (e.g., Naive Bayes [43] and incremental SVM [33]) or no ML model at all (e.g., ETL pipelines). In such scenarios, our materialization and reuse algorithms provide similar absolute performance improvement for both data processing and model training.

7.5 Optimizations Impact

Impact of the Reuse and Plan Generation Algorithm. In this experiment, we analyze the impact of our reuse algorithm on the run time. We run the ensemble use cases in two scenarios. In the first scenario, we compare our system and TFX without any materialization (shown as NoMat in Figure 13). Figures 13a, 13b, and 13c show the impact of our DAG-based reuse algorithm. Sharing the operators of the pipelines alone can increase the performance by a factor of 2 to 3 when compared to TFX. This is because, in the ensemble use cases, the pipelines share similar components. TFX without materialization has to execute each pipeline in isolation with no reuse potential. However, MaR combines the pipelines and executes the similar operators only once. In the second scenario, we compare our system with TFX when the materialization budget is unlimited, i.e., all the artifacts are materialized (shown as FullMat in Figure 13). In such a scenario, we show that MaR outperforms TFX by a factor of 2 to 3, even when the materialization budget is unlimited. In our experiments, we use a limited number of pipelines (six pipelines for Higgs and URL, ten pipelines for Taxi). The addition of new pipelines increases the run time of TFX linearly. However, depending on how similar the pipelines are, MaR can outperform TFX by a larger margin (we also confirmed this in Figure 10).

Optimization Overhead. In our use cases, we use a maximum of 10 pipelines. In these cases, the overhead of the materialization and reuse algorithm was less than 10 milliseconds, which can be contributed to the small number of parallel pipelines. To investigate the overhead of our optimizations, we simulated several workloads of up to 100 pipelines. For every pipeline, we randomly select the number of components from the interval [4, 12] (We found this to be a common number based on the pipelines we observed in the Kaggle competition). Figure 13d shows the overhead of the materialization and reuse algorithm as a function of the number of parallel pipelines. Since we randomly generate the pipelines, we repeat the experiment 10 times and report the average (with error). Figure 13d shows that even with 100 parallel pipelines, our materialization and reuse algorithms incur an overhead of 600 and 100 milliseconds, respectively. In our experiments, pipelines have run times between 500 to 5000 seconds. Therefore, even with a large number of pipelines, our optimizations generate a negligible overhead of less than 0.01%.

8 CONCLUSION

We present a system for optimizing the execution of ML pipelines on overlapping data intervals. We design a materialization algorithm that assigns a benefit score to the generated artifacts of the pipeline operators. The materialization algorithm stores the artifacts with the largest benefit in the artifact store. The benefit score directly correlates to the reduction in the execution cost. Furthermore, we propose a DAG-based reuse and execution plan generation algorithm that considers both the set of materialized artifacts and the set of all the scheduled pipelines. The reuse algorithm generates a plan that minimizes the execution cost by both loading the materialized artifacts and sharing the results of similar operators of different pipelines. We experimentally show that our system can reduce the execution cost of ML pipelines up to an order of magnitude and seamlessly adapt to the changes in the data distribution.

Acknowledgements. This work was funded by the German Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (01IS18025A and 01IS18037A) and the German Federal Ministry for Economic Affairs and Climate Action, Project "ExDRa" (01MD19002B).

REFERENCES

[1] Jordan T Ash and Ryan P Adams. 2019. On the difficulty of warm-starting neural network training. *arXiv preprint arXiv:1910.08475* (2019).

[2] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. 2014. Searching for exotic particles in high-energy physics with deep learning. *Nature communications* 5, 1 (2014), 1–9.

[3] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, et al. 2017. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1387–1395.

[4] Souvik Bhattacherjee, Amit Chavan, Silu Huang, Amol Deshpande, and Aditya Parameswaran. 2015. Principles of dataset versioning: Exploring the recreation/storage tradeoff. In *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*, Vol. 8. NIH Public Access, 1346.

[5] Jeffrey Chung. 2020. *NYC Taxi Trip - Public.* Retrieved August 23, 2021 from kaggle.com/jeffreycbw/nyc-taxi-trip-public-0-37399-private-0-37206

[6] Google Cloud. 2020. *MLOps: Continuous delivery and automation pipelines in machine learning.* Retrieved November 16, 2020 from https://cloud.google.com/solutions/machine-learning/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning

[7] Daniel Crankshaw, Peter Bailis, Joseph E Gonzalez, Haoyuan Li, et al. 2014. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. *arXiv preprint arXiv:1409.3809* (2014).

[8] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A low-latency online prediction serving system. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 613–627.

[9] Nilesh N Dalvi, Sumit K Sanghai, Prasan Roy, and S Sudarshan. 2003. Pipelining in multi-query optimization. *J. Comput. System Sci.* 66, 4 (2003), 728–762.

[10] A. P. Dawid. 1984. Present Position and Potential Developments: Some Personal Views: Statistical Theory: The Prequential Approach. *Journal of the Royal Statistical Society. Series A (General)* 147, 2 (1984), 278–292. http://www.jstor.org/stable/2981683

[11] Jeffrey Dean and Sanjay Ghemawat. 2010. MapReduce: a flexible data processing tool. *Commun. ACM* 53, 1 (2010), 72–77.

[12] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Ziawasch Abedjan, Tilmann Rabl, and Volker Markl. 2020. Optimizing Machine Learning Workloads in Collaborative Environments. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020).

[13] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Tilmann Rabl, and Volker Markl. 2019. Continuous Deployment of Machine Learning Pipelines.. In *EDBT*. 397–408.

[14] Yadolah Dodge. 2008. *Kolmogorov–Smirnov Test.* Springer New York, New York, NY, 283–287. https://doi.org/10.1007/978-0-387-32833-1_214

[15] Iman Elghandour and Ashraf Aboulnaga. 2012. ReStore: reusing results of MapReduce jobs. *Proceedings of the VLDB Endowment* 5, 6 (2012), 586–597.

[16] Yuan Yu Michael Isard Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, and Pradeep Kumar Gunda Jon Currey. 2009. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. *Proc. LSDS-IR* 8 (2009).

[17] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, et al. 2019. Auto-sklearn: efficient and robust automated machine learning. In *Automated Machine Learning*. Springer, Cham, 113–134.

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning.* MIT press.

[19] Nirmal Govind. 2017. *A/B Testing and Beyond: Improving the Netflix Streaming Experience with Experimentation and Data Science.* Retrieved June 08, 2021 from https://netflixtechblog.com/a-b-testing-and-beyond-improving-the-netflix-streaming-experience-with-experimentation-and-data-5b0ae9295bdf

[20] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, et al. 1997. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery* 1, 1 (1997), 29–53.

[21] Michael Greenwald and Sanjeev Khanna. 2001. Space-efficient online computation of quantile summaries. *ACM SIGMOD Record* 30, 2 (2001), 58–66.

[22] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, et al. 2010. Nectar: Automatic Management of Data and Computation in Datacenters. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 75–88.

[23] Charles R. Harris, K. Jarrod Millman, St'efan J. van der Walt, Ralf Gommers, et al. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[24] Sona Hasani, Faezeh Ghaderi, Shohedul Hasan, Saravanan Thirumuruganathan, Abolfazl Asudeh, Nick Koudas, and Gautam Das. 2019. ApproxML: efficient approximate ad-hoc ML models through materialization and reuse. *Proceedings of the VLDB Endowment* 12, 12 (2019), 1906–1909.

[25] Sona Hasani, Saravanan Thirumuruganathan, Abolfazl Asudeh, Nick Koudas, and Gautam Das. 2018. Efficient construction of approximate ad-hoc ML models through materialization and reuse. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1468–1481.

[26] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, et al. 2014. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*. 1–9.

[27] Ihab F Ilyas and Xu Chu. 2019. *Data cleaning.* ACM.

[28] Kaggle. 2020. *New York City Taxi Trip Duration Competition.* Retrieved September 10, 2021 from https://www.kaggle.com/c/nyc-taxi-trip-duration/

[29] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. 2019. AJoin: ad-hoc stream joins at scale. *Proceedings of the VLDB Endowment* 13, 4 (2019), 435–448.

[30] Bojan Karlaš, Matteo Interlandi, Cedric Renggli, Wentao Wu, Ce Zhang, Deepak Mukunthu Iyappan Babu, Jordan Edwards, Chris Lauren, Andy Xu, and Markus Weimer. 2020. Building continuous integration services for machine learning. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2407–2415.

[31] Kieran Kavanagh, David Nigenda, and Aakash Pydi. 2020. *A/B Testing ML models in production using Amazon SageMaker.* Retrieved February 26, 2021 from https://aws.amazon.com/blogs/machine-learning/a-b-testing-ml-models-in-production-using-amazon-sagemaker/

[32] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. 2006. On-the-fly sharing for streamed aggregation. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 623–634.

[33] Pavel Laskov, Christian Gehl, Stefan Krüger, and Klaus-Robert Müller. 2006. Incremental support vector learning: Analysis, implementation and applications. *Journal of machine learning research* 7, Sep (2006), 1909–1936.

[34] Justin Ma, Lawrence K Saul, Stefan Savage, and Geoffrey M Voelker. 2009. Identifying suspicious URLs: an application of large-scale online learning. In *Proceedings of the 26th annual international conference on machine learning*. 681–688.

[35] Imene Mami and Zohra Bellahsene. 2012. A survey of view selection methods. *ACM SIGMOD Record* 41, 1 (2012), 20–29.

[36] Nathan Marz and James Warren. 2013. *Big Data: Principles and best practices of scalable real-time data systems.* Manning.

[37] Ahmed Mazen and Motaz Saad. 2021. *NYC Taxi.* Retrieved August 23, 2021 from kaggle.com/ahmedmurad1990/nyc-taxi

[38] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, et al. 2016. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.

[39] Quentin Monmousseau. 2019. *ML Workflow.* Retrieved August 23, 2021 from kaggle.com/quentinmonmousseau/ml-workflow-lightgbm-0-37-randomforest-0-39

[40] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. 2010. MRShare: sharing across multiple queries in MapReduce. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 494–505.

[41] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[42] Cedric Renggli, Bojan Karlaš, Bolin Ding, Feng Liu, Kevin Schawinski, Wentao Wu, and Ce Zhang. 2019. Continuous Integration of Machine Learning Models with ease.ml/ci: Towards a Rigorous Yet Practical Treatment. In *Proceedings of Machine Learning and Systems*, A. Talwalkar, V. Smith, and M. Zaharia (Eds.), Vol. 1. 322–333.

[43] Irina Rish et al. 2001. An empirical study of the naive Bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, Vol. 3. 41–46.

[44] Prasan Roy, Srinivasan Seshadri, S Sudarshan, and Siddhesh Bhobe. 2000. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 249–260.

[45] Sebastian Schelter, Felix Biessmann, Tim Januschowski, David Salinas, Stephan Seufert, and Gyuri Szarvas. 2018. On challenges in machine learning model management. (2018).

[46] Timos K Sellis. 1988. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)* 13, 1 (1988), 23–52.

[47] Ben Sharma. 2018. *Architecting data lakes: data management architectures for advanced business use cases.* O'Reilly Media.

[48] Jaeho Shin, Sen Wu, Feiran Wang, Christopher De Sa, Ce Zhang, and Christopher Ré. 2015. Incremental knowledge base construction using deepdive. In *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, Vol. 8. NIH Public Access, 1310.

[49] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. 2015. General incremental sliding-window aggregation. *Proceedings of the VLDB Endowment* 8, 7 (2015), 702–713.

[50] NYC Taxi and Limousine Commission (TLC). 2021. *NYC Taxi and Limousine Commission (TLC) Trip Record Data.* Retrieved August 23, 2021 from https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

[51] Jonas Traub, Philipp M Grulich, Alejandro Rodríguez Cuéllar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. 2019. Efficient Window Aggregation with General Stream Slicing. In *EDBT*. 97–108.

[52] Cheng-Hao Tsai, Chieh-Yen Lin, and Chih-Jen Lin. 2014. Incremental and decremental training for linear classification. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 343–352.

[53] Tom van der Weide, Dimitris Papadopoulos, Oleg Smirnov, Michal Zielinski, and Tim van Kasteren. 2017. Versioning for end-to-end machine learning pipelines. In *Proceedings of the 1st Workshop on Data Management for End-to-End Machine Learning*. 1–9.

[54] Manasi Vartak, Joana M F. da Trindade, Samuel Madden, and Matei Zaharia. 2018. Mistique: A system to store and query model intermediates for model diagnosis. In *Proceedings of the 2018 International Conference on Management of Data*. 1285–1300.

[55] Guoping Wang and Chee-Yong Chan. 2013. Multi-query optimization in mapreduce framework. *Proceedings of the VLDB Endowment* 7, 3 (2013), 145–156.

[56] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya Parameswaran. 2018. Helix: Holistic optimization for accelerating iterative machine learning. *arXiv preprint arXiv:1812.05762* (2018).

[57] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 15–28.

[58] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. https://doi.org/10.1145/2934664

[59] Ce Zhang, Arun Kumar, and Christopher Ré. 2016. Materialization optimizations for feature selection workloads. *ACM Transactions on Database Systems (TODS)* 41, 1 (2016), 1–32.