A Practical Cross-Layer Approach for ML-Driven Storage Placement in Warehouse-Scale Computers

Abstract

Storage systems account for a major portion of the total cost of ownership (TCO) of warehouse-scale computers, and thus have a major impact on the overall system's efficiency. Machine learning (ML)-based methods for solving key problems in storage system efficiency, such as data placement, have shown significant promise. However, there are few known practical deployments of such methods. Studying this problem in the context of real-world hyperscale data center deployments at *AnonCorp*, we identify a number of challenges that we believe cause this lack of practical adoption. Specifically, prior work assumes a monolithic model that resides entirely within the storage layer, an unrealistic assumption in real-world data center deployments. Additional challenges include adaptability, reliability, and interpretability.

We propose a cross-layer approach that moves ML out of the storage system and performs it in the application running on top of it, co-designed with a scheduling algorithm at the storage layer that consumes predictions from these application-level models. This approach combines small, interpretable models with a co-designed heuristic that adapts to different online environments.

We build a proof-of-concept of this approach in a production distributed computation framework at *AnonCorp*. Evaluations in a test deployment and large-scale simulation studies using production traces show improvements of as much as 2.48× in TCO savings compared to state of the art baselines. We believe this work represents a significant step towards more practical ML-driven storage placement in warehouse-scale computers.

Keywords: Machine Learning for Systems, Data Placement Optimization, Data Centers, Storage Systems

1 Introduction

Storage systems comprise a large part of data centers' total cost of ownership (TCO). Even small improvements in storage system efficiency can have a major impact on the overall costs. Improvement as low as 1% represents a large amount in the context of hyperscale data centers, which see billions of dollars of investment [25]. Data placement – e.g., deciding whether a file should be stored on hard disk (HDD) or solid state drives (SSD) – is an important decision impacting the efficiency and costs in data center storage systems. This problem is also known as storage tiering and has been the subject of a large amount of research [8, 18, 30].

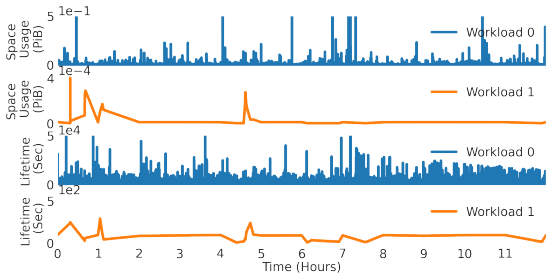

Figure 1. Individual workloads characteristics: Workloads share drastically different resource usage patterns.

In this paper, we focus on a particular instance of this problem: placement of intermediate files in data processing frameworks such as Apache Beam [26] or Apache Spark [40]. These frameworks consume data sets that can reach petabytes in size [34] and perform large-scale computations on this data. These computations move data between a large number of servers and repeatedly materialize data as *intermediate files*. These intermediate files themselves alone can account for a significant portion of storage resources in a data center (up to 35% in some clusters). Currently, there are two approaches to the tiering problem in these frameworks:

- Heuristics, such as greedily allocating data to SSDs until capacity is reached and then using HDDs for overflow [3, 9, 35, 37, 38]. These heuristics are deployed and represent today's state-of-the-art. They are fast and interpretable, but perform suboptimally when SSD capacity is limited.
- Machine Learning (ML) approaches that learn and leverage real-world workload information [42]. Few of these approaches are practically deployed due to considerations such as run-time overhead, adaptability, or decomposition challenges, and risks associated with a model becoming a single point of failure [23, 28].

To understand the challenges behind adoption of ML in real-world scenarios, we analyzed our real-world production systems at *AnonCorp*. Data centers run a wide range of workloads with vastly different characteristics (Figure 1). Workloads arrive and evolve at a high rate, and data access patterns are highly dynamic and application-specific. Data centers deal with this issue through *abstraction layers*, such as the application, storage or hardware layer: For example, the application layer does not need to worry about the specifics of the hardware, and the storage layer does not need to know any details about the inner workings of each

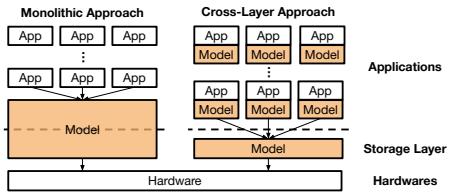

Figure 2. Conceptual overview of the monolithic approach vs. the cross-layer approach.

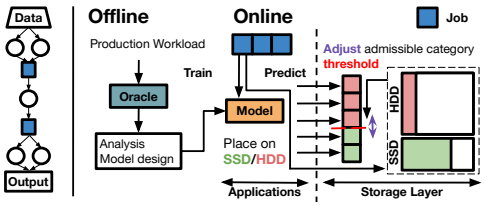

Figure 3. Left: Data flow graph in a data processing framework. Data is processed in parallel and its jobs create intermediate files (blue) which are inputs for the next processing step. **Right:** Approach Overview. We analyze production workloads offline for model design and training. Online, each application's model predicts job properties and passes the prediction to the storage layer for job placement.

application. This enforces separation of concerns and allows these components to operate and evolve independently.

We posit that this is the key challenge behind existing proposed ML methods. Existing ML works mostly treat the end-to-end data placement as one problem and assume a monolithic model deployed within the storage system [15, 22, 32, 42]. Such a model might be trained on file names or common application behavior [42]. While this approach works in simulation, it breaks the separation of concerns, which is problematic in real-world large-scale systems. For instance, changes to a major application that, e.g., affect file names would need to trigger a retraining of the model at the storage layer, which needs to roll out behavior changes at a much lower velocity than the workloads.

Our design embraces the multi-layer characteristic of storage systems and presents a practical solution to these problems by combining cheap and interpretable ML models at the application layer with a custom algorithm that leverages their predictions at the storage layer. Instead of a single large ML model, we build smaller models for individual workloads, which produce hints that the storage layer can utilize to place the workload's data (Figure 2).

We first present a headroom analysis to fully understand the potential upside from ML over traditional heuristics. We formulate the data placement problem into an Integer Linear Programming (ILP) problem and use a solver to determine optimal placement decisions for each job. We find that these optimal decisions from an ILP solver can achieve 3.44× the cost savings of a state-of-the-art heuristic approach (but require clairvoyant knowledge).

Prior work has proposed imitation learning against such an oracle [22] which assumes a monolithic model for data placement in storage systems. However, we find that this approach does not work in our deployment, since the model does not only need to make decisions for individual workloads but adapt to an environment that is changing due to external factors (e.g., varying load patterns and workloads arriving or leaving). We thus devise a *cross-layer* approach to tackle this *adaptability problem*. At the application layer, we analyze the data properties that contribute to the optimal placement. We then design a category model to predict the ranking of these properties, which is independent of online fluctuations of the environment and other applications. Finally, we design an adaptive algorithm at the storage layer to select the data to place on SSD based on the model predictions from all applications and system feedback.

We demonstrate our approach in the context of a real production data processing framework at *AnonCorp*. We show its practicality by developing a prototype of our approach and running it in a test deployment. We also perform an extensive simulation study based on real-world production traces from *AnonCorp*'s data centers. We show that our approach can lead to an additional 3.2% TCO savings, more than 2.48× the savings from the production baseline. We summarize our contributions as:

- Within a real-world production setup, we investigate ML for storage placement from a new perspective, with a focus on practicality in production settings.
- We design and implement a novel cross-layer approach, combining ML and heuristics that can adapt across workloads and external factors in data centers.
- We prototype the proposed ML integration to show its realizability in a real production codebase.
- We evaluate our method at scale with real production traces and achieve 2.48× TCO savings compared to state-of-the-art baselines.

We first present background on storage systems and production constraints (Section 2), and formulate our optimization problem with baselines (Section 3). We then discuss our main approach, introducing our ML method design (Section 4.2) and scheduling algorithm (Section 4.3). We next show detailed prototype and large-scale simulation studies (Section 5) of our approach. Finally, we present related work (Section 6) and conclude (Section 7).

2 Background

2.1 Storage for Data Processing Frameworks

Modern data processing frameworks, such as Apache Beam, structure their computations as data flow graphs (Figure 3,

left). Each node (or *step*) within the graph represents a computation step. Edges represent the flow of data. Computations are highly parallel, and a distributed framework spawns *workers* to execute tasks. A worker is a process that runs on a server. Data is generally passed between workers through *shuffle jobs*. A shuffle job is generated when the execution of the workflow reaches a step or operation that necessitates the exchange of information. As an example, *GroupByKey* is a common operation across frameworks that generates one or more shuffle jobs. During a shuffle job, workers write their data as *intermediate files* to a distributed file system [4, 10, 31] and read it in subsequent steps. The access patterns to the files depend on the specifics of the computation, such as filtering, grouping, or sorting. One shuffle job can operate on multiple intermediate files.

2.2 SSD/HDD Tiering and its Trade-Offs

Storage cost falls into several different categories: 1) the amount of storage (e.g., in GiB) occupied by the data, 2) wearout of devices such as SSD, which degrade with every write, and 3) the amount of I/O operations (i.e., read or write requests to a storage device per unit time sustained by the device). SSDs and HDDs have different trade-offs among all three dimensions. SSDs provide much larger amounts of I/O with a higher cost per GiB and write-induced wearout. In contrast, HDDs are ideal for large amounts of data and long sequential access patterns that introduce few I/O operations. At the same time, SSDs are ideal for random, small accesses – if the resulting wearout can be tolerated. In practice, intermediate files in data processing pipelines can fall into either category (or, more commonly, in between), which makes the data placement problem challenging.

2.3 Production Requirements and Limitations

Data centers accommodate a vast array of workloads with diverse behavior patterns. Employing a single model to jointly learn all workloads introduces a single point of failure, and requires the model to be large and complex, and thus expensive and difficult to interpret. Further, this approach requires all input features be reliably delivered to the storage system, when some of the most predictive features are workload-specific [42]. Finally, In hyperscale data centers, workloads exhibit significantly faster rates of change compared to the deployment and update cycles of storage systems. This causes a dilemma: 1) Rolling out the model with the storage system means it is stale by the time it reaches production; 2) updating the model independently of the storage system means that it is not tested as rigorously as the rest of the system, becoming its weakest link.

To address these problems, we propose a more granular approach where each workload has its own dedicated model to produce a *hint*, which is then reliably passed to the storage system. This hint indicates, for example, how well a file can be cached. Since workloads "*bring their own model*", models evolve at the velocity of the workload rather than the storage system. Because the models are smaller, they are cheaper and more interpretable. They are distributed across many workloads hence they can use more features and are more robust: a model failure only affects one workload. In this work, we focus on optimizing data placement for data processing pipelines, but our cross-layer design is general.

Training a model offline and deploying it online is challenging, since a static model cannot adapt to evolving workload patterns, which are prevalent in real-world scenarios. To address this issue, we present an adaptive strategy that utilizes online observations to inform placement decisions.

Models can introduce non-trivial overheads. For example, prior work suggested using Transformers, which are known for their superior learning capabilities but can be computationally expensive, incurring prediction costs of approximately 99ms per prediction [42]. To balance performance and efficiency, we leverage gradient boosted trees as the learning model, providing a compromise between lightweight, low-performance models and powerful, expensive models.

3 Problem Formulation and Baselines

We now discuss the data placement problem concretely. Our basic data placement unit is a shuffle job. A program can have from 0 to hundreds of shuffle jobs. We track four attributes for each job: (start time, lifetime, job size, cost). Job size is measured in bytes. A cluster has an SSD capacity. The placement algorithm produces a mapping: job $\rightarrow$ {SSD, HDD}.

Performance is measured in savings. First, we measure how much HDD I/O can be reduced. Moving jobs to SSDs can free up HDD for I/Os that cannot be moved away from HDD (such as accesses of cold data). We quantify the savings in HDD I/O through a metric we call *Total Cost of I/O* (TCIO), where 1.0 is the amount of I/O that a representative hard drive can sustain per second. TCIO of jobs on SSDs is zero.

Second, we also measure monetary savings. For simplicity, we only consider the change in storage costs and ignore costs from other parts of the job (CPU, RAM, etc.) We define *Storage Total Cost of Ownership* (simply referred to as TCO from now on), which covers the total expenditure in dollars associated with acquiring, operating, and maintaining a storage system. The TCO for HDD and SSD are defined differently due to the nature of devices. Substitute Dev for HDD or SSD below to get the respective definition:

$$\begin{aligned}
\text{TCO}^{\text{Dev}} &= \text{cost}^{\text{Dev}}_{\text{byte}} + \text{cost}^{\text{Dev}}_{\text{network}} + \text{cost}^{\text{Dev}}_{\text{server}} + \text{cost}^{\text{Dev}}_{\text{specific}} \\
\text{cost}^{\text{Dev}}_{\text{byte}} &= \text{byte_cost}^{\text{Dev}} \cdot \text{size} \cdot \text{duration} \\
\text{cost}^{\text{Dev}}_{\text{network}} &= \text{network_cost_rate} \cdot \text{IO_throughput}^{\text{Dev}} \\
&\quad \cdot \text{duration} \\
\text{cost}^{\text{HDD}}_{\text{server}} &= \text{server_cost_rate}^{\text{HDD}} \cdot \text{TCIO} \cdot \text{duration} \\
\text{cost}^{\text{SSD}}_{\text{server}} &= \text{server_cost_rate}^{\text{SSD}} \cdot \text{write_throughput}
\end{aligned}$$

$$\text{cost}^{\text{HDD}}_{\text{specific}} = \text{device\_cost\_rate}^{\text{HDD}} \cdot \text{TCIO} \cdot \text{duration}$$

$$\text{cost}^{\text{SSD}}_{\text{specific}} = \text{wearout\_cost\_rate}^{\text{SSD}} \cdot \text{total\_written\_bytes}$$

where $*$_cost_rate denotes conversion rates to dollar cost; $\text{cost}^{\text{Dev}}_{\text{byte}}$ denotes the cost of storing one byte for one second on a device; TCIO, size, and duration denote a job's TCIO need, storage footprint, and duration (for example, if a job has a TCIO of 2, the job would need two HDDs to run). The $\text{cost}^{\text{Dev}}_{\text{network}}$ is a value derived from the data center total network cost of transmitting data at the maximum throughput divided by the throughput per second. $\text{cost}^{\text{HDD}}_{\text{server}}$ and $\text{cost}^{\text{HDD}}_{\text{specific}}$ cover the cost of the servers and HDDs.

In our practice, we found that the server cost for running a job on SSD correlates with the bytes transmitted. Because all SSD devices have a limit on the amount of Program/Erase (P/E) cycles and each write causes a loss in monetary value, the $\text{cost}^{\text{SSD}}_{\text{specific}}$ is included to cover these wearout costs. $\text{wearout\_cost\_rate}^{\text{SSD}}$ is calculated from the specific SSD drive model's total bytes written rating.

Oracle: Optimal Solution Based on Solver. To better understand the headroom that is available if we achieve perfect data placement, we employ an oracle. It is an upper bound on the best solution, but impossible to implement. The oracle policy is based on using an Integer Linear Programming (ILP) solver by assuming clairvoyant knowledge — that is, assuming we know the future access pattern.

We formulate the placement problem as an ILP problem. The SSD space limit is M and we assume the HDD space is infinite due to its lower cost per GiB. $X = \langle x_0, \ldots, x_N \rangle$ is a sequence of arriving jobs. Job i, represented by variable x_i, arrives at time a_i, ends at time e_i, and it needs s_i space with c_i^{SSD} cost to put on SSD, c_i^{HDD} cost to put on HDD. Oracle optimization can either optimize for TCIO or TCO. We use a binary variable, x_i, to denote the data placement decision, $x_i = 1$ if i is put on SSD, and $x_i = 0$ if i is put on HDD. Once placed, a job x_i runs from a_i to e_i and $p_i(t) = x_i s_i$ represents the job's SSD consumption at time $t \in [a_i, e_i]$. Now the problem becomes maximizing the savings by placing jobs on SSD under space constraints (SSD space is limited):

$$\begin{aligned}
& \max \sum_{i \in l} x_i (c_i^{\text{HDD}} - c_i^{\text{SSD}}) \\
\text{subject to:} \quad & x_i \in \{0, 1\}, \forall i \in [0, N] \\
& p_i(t) = x_i s_i, \forall i \in [0, N], a_i \le t \le e_i \\
& \sum_{i \in [0,N], a_i \le t \le e_i} p_i(t) \le M, \forall t \in T \\
& T = \max\{e_i : i \in [0, N]\}
\end{aligned}$$

We run the above ILP with the historical production workload data and find the optimal placement decisions that save the maximum amount of cost. The solver is optimal because it has clairvoyant knowledge. The clairvoyant knowledge includes information that is not available at a job's placement decision time in practice: 1) The solver knows the global job ranking in terms of cost savings and would prioritize putting high cost saving jobs onto SSD. 2) The solver knows the workload patterns and places jobs that would monopolize SSD resources on HDD instead.

In addition to clairvoyant knowledge, the oracle needs a fixed SSD capacity limit for optimal placement. However, we do not have such knowledge ahead of time as the data center is shared between many jobs (not just data pipelines) and free SSD capacity fluctuates over time. Thus, a solution that can apply under varying SSD capacities is needed.

FirstFit: Static Placement. Production systems commonly perform HDD/SSD tiering using FIFO or LRU-style heuristics [9, 36]. We implement a representative instance of such a heuristic. We try to place jobs on SSD in the order of their start times. This approach optimizes TCIO when unlimited SSD is available but can significantly increase TCO when SSD capacity is limited or expensive. This baseline checks a job's peak space usage and decides only to place jobs on SSD that fit the SSD availability.

Heuristic: Practical Adaptive Placement. Recently, heuristics that can dynamically adapt to workloads have been introduced, striking a balance between dynamically learning behavior and avoiding the practicality issues in Section 2.3. We emulate the state-of-the-art placement approach from [38]. It focuses on a slightly different problem (SSD read cache admission), but can be adapted for our placement task. We use this approach as a stand in for the closest practical approach to a learning-based baseline.

The approach starts from a set of categories associated with storage requests and then constructs a per-category admission policy based on dynamic behavior. In our experiments, we use the job's *ID* as the category. For each job category, the approach measures space usage and TCO savings. We rank the categories by their TCO savings and add categories into an *admission set* until the selected category's historical space usage reaches the SSD capacity. When a new job arrives, we decide to place it on SSD if it belongs to the admission set. Otherwise, the job is placed on HDD.

ML Baseline: Lifetime Prediction for SSD / HDD Tiering. We include another closely related ML-driven approach [42], which models storage problems in data centers as distribution prediction problems. We follow the SSD/HDD tiering case study in [42] to predict the mean (μ) and standard deviation (σ) of a file lifetime. Files with a predicted lifetime ($\mu+\sigma$) shorted than the specified time-to-live (TTL) are admitted to SSD. To mitigate potential mispredictions, we evict any file residing in the SSD for longer than $\mu + \sigma$ following [42].

4 Hybrid Learning Approach

One frequent approach to ML-driven systems is to train a model that learns to make decisions, such as whether to place a file on SSD or HDD – e.g., via imitation learning [22]. However, data centers are highly dynamic environments and

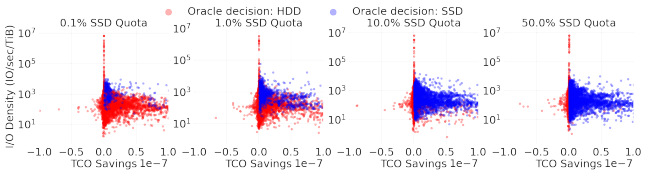

Figure 4. I/O density and TCO savings of each job (color shows oracle placement decision when optimizing for TCO). Tested under different SSD quota.

the optimal decision depends on external factors such as the available amount of SSD at a given point in time. Thus, a model would need to jointly learn the external environment *and* the workload, which is challenging and not deployable, requires a possibly prohibitive amount of training data, and may be brittle when encountering new scenarios.

To address this and our other challenges (Section 2.3), we propose a *cross-layer* learning approach that uses the model only to predict a *proxy* for workload-specific characteristics and then co-designs a storage-level heuristic to turn the predictions into decisions for the current environment.

Specifically, we want to design a proxy that correlates with one job's TCO savings. The proxy allows predicted results to directly represent how a job's placement contributes to end-to-end cost savings. We define such a contribution as "importance" and train a categorical pointwise ranking model (*category model*) to learn the job's *importance ranking* (Figure 3 right). Each category maps to a set of importance rankings. A higher ranking indicates a more important job – placing it on SSD saves more cost. When making a placement decision, we query the model for an *importance ranking category* for a new job. We then run an adaptive category selection algorithm with dynamic feedback from the storage layer to decide the admissible importance ranking categories onto SSD and the predicted category's admissibility.

4.1 Features

We train our model on application-level features from production traces. The features span execution metadata, job timestamps, allocated resources, and historical system metrics, which reflect how jobs are processed in our cloud center.

At a high level, a job comprises three steps. We assume that each worker possesses a number of data records in the working memory. In the first step, each worker writes the data it owns into raw intermediate files. Accessing the data in these raw files is inconvenient because they lack a specific order. To address this issue, sorters organize the data records in these files into sorted intermediate files as part of the second step. Thirdly, the workers retrieve their required data from the sorted intermediate files back into their working memory, concluding the job. If feasible, these three steps can be executed concurrently, resulting in temporal overlap.

The I/O density of jobs depends on how these data records are written and read, so we provide as much internal job-related information from the framework to the model as possible. Internally, the data a workflow needs to process is divided into buckets. A bucket is a unit of work that is assigned to a worker. Each bucket contains a set of tasks that are executed by a single worker. The number of buckets is determined by the data to be shuffled and the number of workers available. Buckets are used to ensure that work is distributed evenly across workers and that no worker is overloaded.

In the first step of a job, the worker shards the data in each bucket into shards, and each shard is assigned to a writer for being written to storage. A writer packs data into stripes and writes one stripe at a time. This enables parallel writing and faster write throughout.

The feature we choose (Table 1) reflects how these steps are executed (allocated resources, execution metadata). Execution metadata is formatted as strings that detail the execution path and location. Key elements are separated by non-alphanumeric characters. We treat execution metadata as a sequence of substrings representing the key elements (Table 2). Since executions may run periodically, we also include the weekday and hour of the day of a job's start time. The allocated resource information represents resources assigned to the job by the cluster scheduler, before it starts execution. However, specific details regarding resource distribution, such as the assignment to SSD or HDD, are not determined at this stage. In addition, we also incorporate properties of previously completed jobs from the same user's pipelines, including the past TCIO, job lifetime, and size.

4.2 Model Design

We use gradient boosted trees instead of neural networks (which are much more expensive and less interpretable) or lookup tables (which sacrifice accuracy). We build on the Yggdrasil Decision Forests framework [11]. Our model is trained on features in Section 4.1.

Quantifying Job Importance. Our goal is for the model to determine each job's *importance*, which is equivalent to the expected cost savings. We first want to design a way to represent the job *importance*. We examine the oracle placement (from Section 3) under different SSD capacities. We expect the most important jobs to be admitted by the oracle even under extremely limited SSD capacity, and as the SSD capacity increases, less important jobs are admitted.

For each job, we compute a binary placement decision (SSD/ HDD) from the oracle with different SSD capacity limits. In Figure 4, we show how oracle decisions correlate with TCO savings and I/O density, which denotes the total I/O across the job lifetime divided by its maximum storage footprint. As the SSD capacity increases, more jobs with lower I/O density are chosen for SSD. Since the oracle optimizes for TCO savings, jobs with negative TCO savings if put on

Features	Feature Group	Description
average_TCIO	Historical system metrics	Average TCIO of the job's historical executions.
average_size	Historical system metrics	Average peak intermediate file size of the job's historical executions.
average_lifetime	Historical system metrics	Average job historical lifetime.
average_I/O density	Historical system metrics	Average I/O density of the job's historical executions.
bucket_sizing_initial_num_stripes	Allocated resources	The initial number of stripes a shard is expected to be divided into. Each stripe contains a couple of data records.
bucket_sizing_num_shards	Allocated resources	The number of shards the working set is expected to be sharded into.
bucket_sizing_num_worker_threads	Allocated resources	Number of worker threads.
bucket_sizing_num_workers	Allocated resources	Number of workers in this job.
initial_num_buckets	Allocated resources	The initial number of buckets the job uses when it was started.
num_buckets	Allocated resources	The number of buckets the current job actually uses.
records_written	Allocated resources	The number of records to be shuffled for a shuffle job.
requested_num_shards	Allocated resources	Number of shards the current working set is requested to be sharded into.
open_time_dayhour	Job timestamp	The hour of the job start time.
open_time_seconds	Job timestamp	The second of the job start time.
open_time_weekday	Job timestamp	The week day of the job start date.
build_targetname	Execution metadata	The target in the build file that is used to build the executable binary.
execution_name	Execution metadata	A user-assigned identifier for the job. Usually set to the binary file name.
pipeline_name	Execution metadata	Name of the pipeline the job belongs to. A pipeline contains multiple jobs.
step_name	Execution metadata	A computer generated step identifier from the workflow's execution graph.
user_name	Execution metadata	Name of the workflow step that is starting the shuffle job.

Table 1. Feature details.

Features	Example Values
build_targetname	//storage/[redacted]/build_manager:[redacted]
execution_name	com.[redacted].[redacted].[redacted].trigger2.launcher.Main
pipeline_name	[redacted]_org_[redacted]_indicator_metrics.
	[redacted]-dims_prod.[redacted].data_importer
step_name	[redacted]-open-shuffle10
user_name	GroupByKey-22

Table 2. Execution metadata feature examples.

SSD should never be selected. Further, if two jobs have the same I/O, short-lived and small-sized jobs are preferred as less SSD capacity and usage time is occupied. This suggests that predicting the *sign of TCO savings* and *I/O density* is a good proxy for job importance: negative TCO saving jobs are least important; jobs with higher I/O density are more important.

Label Design. Predicting precise values of these properties has been shown to be challenging, even in theoretical works – e.g., lifetime in [42]. Rather than treating the importance prediction problem as regression, we choose a categorical pointwise ranking model, which groups jobs with similar (TCO savings, I/O density) into the same category – same importance ranking class. The idea of framing the output prediction issue as a classification task is commonly adopted in the fields of image and audio analysis [17, 21].

To pick these specific categories, we need to take into account that our goal is for these categories to provide a ranking of the "importance" of placing each job on SSD. First, negative TCO saving jobs should have the lowest ranking and we set aside one category specifically for these jobs. For the remaining categories, our goal is to cluster jobs by their I/O density. We found that both linear and logarithmically spaced categories would result in a heavily imbalanced data set (Figure 4). Therefore, we choose the categories so that they evenly divide the training set by I/O density (e.g., top 10%, top 20%, top 30%, etc.) For a model with N categories, this results in the following category labels, given TCO savings $x.m$, I/O density $x.n$, and training set size D:

$$\mathrm{C(x)} = \begin{cases} 0, & \text{if } x.m < 0 \\ k, & \text{if } x.n \in (\text{top } \frac{N-k}{N-1} * D, \text{top } \frac{N-k-1}{N-1} * D] \\ & \quad \text{and } x.m \geq 0 \end{cases} \tag{1}$$

4.3 Adaptive Category Selection Algorithm

We now discuss how our cross-layer design combines the learned category model and a heuristic-based algorithm for online data placement. As mentioned in Section 4, the model's category prediction is independent of the SSD capacity. How to select the categories to place on SSD under different SSD capacities is unknown. A simple approach is to fix the admittable categories and always only place jobs predicted in these categories onto SSD. However, this strategy can only work when we have a hard constraint about the category admission, e.g. we should never admit negative TCO saving jobs. As shown in Figure 4, when we have larger SSD capacities, we want to admit more jobs compared to the smaller capacities case – that is, more categories.

Algorithm Overview. Our algorithm makes the admission decisions based on real-time feedback regarding SSD utilization. When observing limited SSD capacity, we gradually decrease the number of categories to admit. Otherwise, we admit more categories. Since our category model predicts the "importance ranking" of jobs, admitting fewer categories

Algorithm 1 Adaptive Category Selection Algorithm

input model M_N, $X = \langle x_0, \ldots, x_n \rangle$, t_w, $T_{\text{SpilloverTCIO}}$, t_l.

Initialize $t_d = 0$, ACT = 1 and $X_h = \emptyset$.
for x_i in X **do**
  Get the current time stamp as x's arrival time $t_i = x_i.t_a$
  if Last admission decision is expired: $t_i \geq t_d + t_l$ **then**
    Update look back window endpoint $w_s, w_e = t_i - t_w, t_i$
    Remove expired jobs: $X_h = X_h - \{x_j | x_j.t_a <= w_s\}$
    Update the spillover percentage from X_h:
    $h_{\text{SpilloverTCIO}} = P_{\text{SpilloverTCIO}}(X_h, t_i)$
    if $h_{\text{SpilloverTCIO}} < T_l$ **then**
      ACT = $\max(N-1, \text{ACT}+1)$ **end**
    if $h_{\text{SpilloverTCIO}} > T_u$ **then**
      ACT = $\min(1, \text{ACT}-1)$ **end**
    Update decision making time $t_d = t_i$ **end**
  Infer the predicted category $C_i = M_N(x_i.\text{features})$
  if $C_i \geq$ ACT **then**
    Place x_i onto SSD **else**
    Place x_i onto HDD **end**
  Add the job into the observation history $X_h = X_h \cup x_i$
end for

Symbol/Notation	Meaning
$X = \langle x_0, x_1, \ldots, x_n \rangle$	job sequence
x.features	job features available before execution
x.Dev	job scheduled device (0/1 for HDD/SSD)
$t_a, x.t_a$	job arrival time
$t_s, x.t_s$	job spillover time
$t_e, x.t_e$	job end time
$\text{TCIO}_{\text{HDD}}(t)$	job TCIO if put onto HDD till t
$\text{SpilloverTCIO}(x, t)$	job spillover TCIO at t
$P_{\text{SpilloverTCIO}}(X, t)$	jobs spillover TCIO percent at t
t_w	look back window time length
t_l	admission decision effective time length
$T_{\text{SpilloverTCIO}} = [T_l, T_u]$	spillover tolerance range
t_d	the last placement decision making time
X_h	job observation history
ACT	admission category threshold ($\leq N-1$)
M_N	decision tree model with N categories

Table 3. Algorithm notation.

naturally leads to admitting the most important jobs. Admitting more categories means that we broaden the admission set by adding less important but still cost saving jobs. We use a sliding category admission threshold to determine what part of the predicted categories get placed on SSD.

SSD Usage Approximation. In cloud data centers, the actual SSD capacity varies among clusters, which is also hard to directly approximate in practice. The criteria for determining whether an SSD is nearly full (i.e. can not fit more jobs) or not are influenced by workload patterns. Thus, we introduce a metric to unify the measurement of SSD capacity usage across clusters and workloads through job behavior observation. Given a sequence of jobs, $X = \langle x_0, x_1, \ldots, x_n \rangle$, we introduce *spillover TCIO percentage*, $P_{\text{SpilloverTCIO}}(X, t)$, to measure the portion of all job x_i's TCIO that is scheduled to be put onto SSD but ends up on HDD due to the fact that the SSD has already reached its full capacity at timestamp t:

$$P_{\text{SpilloverTCIO}}(X, t) = \frac{\sum_{x_i \in X} \text{SpilloverTCIO}(x_i, t)}{\sum_{x_i \in X} x_i.\text{Dev} \cdot x_i.\text{TCIO}_{\text{HDD}}(t)}$$

where $\text{SpilloverTCIO}(x, t)$ is the amount of spill over TCIO of a job x at time t. t_s is x spillover time:

$$\begin{cases} \frac{t - t_s}{t - t_a} \text{TCIO}_{\text{HDD}}(t), & \text{if } t_s \text{ exists and } t_a \leq t_s \leq t \\ 0.0, & \text{Otherwise.} \end{cases}$$

Notation is in Table 3. Intuitively, SpilloverTCIO measures the amount of the job's scheduled TCIO savings that are not realized. A large $P_{\text{SpilloverTCIO}}$ means that few jobs can be successfully scheduled onto SSD, which indicates that the SSDs are nearly full.

Algorithm Design. We now introduce the adaptive category selection algorithm in Algorithm 1 with notation available in Table 3. In the algorithm, we keep track of an observation history X_h, which contains all the jobs starting within a fixed look back window, and calculate the SpilloverTCIO within the history, $h_{\text{SpilloverTCIO}}$. Then, we adaptively adjust the admission category threshold (ACT) based on the observed $P_{\text{SpilloverTCIO}}$ – if $P_{\text{SpilloverTCIO}}$ is larger than ACT, we increase the threshold to admit fewer categories. One issue of a dynamic control system of this kind is that ACT may change drastically. We provide two designs to smooth the threshold change:

- We use a spillover tolerance interval, $T_{\text{SpilloverTCIO}}$. When the observed SpilloverTCIO falls into the interval, we maintain ACT. If $P_{\text{SpilloverTCIO}}$ is below the lower bound of $P_{\text{SpilloverTCIO}}$, we decrease the threshold by 1 (to avoid large adjustments). Conversely, for high $P_{\text{SpilloverTCIO}}$, we increase the ACT by 1. This provides more flexibility for ACT adjustments.
- The threshold update is triggered by job arrival and the decision interval t_l seconds instead of simply job coming to decrease the threshold change times.

When we designed the algorithm, we discovered that considering all the jobs *starting* within the look back window can result in a more accurate estimate of the latest SSD usage information than using all the jobs *overlapping* the look back window. We think this could be the result of jobs with a long lifetime having an outsize effect in such a setting.

5 Evaluation

We study the following research questions for evaluation:
RQ1: What is our method's performance when integrated into *AnonCorp*'s system?
RQ2: What are our method's TCO and TCIO savings?
RQ3: What are our method's TCO savings under different SSD space constraints?
RQ4: How does our method generalize across clusters, or perform with new users and pipelines?

RQ5: Is our ML model practical? Which features contribute most to learning?
RQ6: How sensitive is our method under different hyperparameters? What are the important parts of our design?
RQ7: How does the adaptive ranking algorithm look like?

5.1 Experimental Setup

Metrics. We evaluate our performance using two metrics: *TCO savings percentage* and *TCIO savings percentage*. As mentioned in Section 3, the TCO in our work includes the total expenditure of maintaining a storage systems (such as I/O cost, SSD wearout, etc). We present the results with the percentage of TCO savings over the total TCO if all jobs are put on HDD. The TCIO measures the actual I/O cost without calculating the SSD wearout cost. Given that the SSD wearout cost could differ in different contexts, we believe showing TCIO can help understand the savings purely from an I/O perspective. Similar to the percentage of TCO savings, we show the TCIO savings by the percentage of TCIO savings over the TCIO if all jobs are put on HDD.

Data Collection & Model Training. We collect the production traces from *AnonCorp* that consist of the historical execution log of the data processing framework and the I/O traces from the distributed storage system. These traces contain jobs' metadata and post-execution measurements, such as job lifetime and TCIO. Section 4.2 explains the features we use. Our training and test dataset each contains one week's data, which are collected from a contiguous two-week time span. The training data are constructed by joining two sources of data. The first is the job execution record, which includes detailed execution metadata about each job. The second source is the metrics data from the underlying file system, which includes lifetime, size of each job; total amount and mean of read and write I/O ops; and total bytes read and written. We use a 15-class gradient-boosted trees model with 300 trees at maximum and a max depth as 6 for all of our models. We train a separate model for each cluster.

End-to-end System Integration. We develop a prototype and deploy it in a production data processing framework and distributed storage system at *AnonCorp* to understand the practicality of our algorithm. In the prototype, we first follow Section 4.1 to train the per-workload model offline. During execution time, the computation framework collects required features and loads the model to perform inference, to generate a categorization result before opening files for writing. The categorization results are passed to the storage cache server, which makes real-time decisions for placement on HDD or SSD. Metrics are collected during the process to evaluate TCO and TCIO. We collect the following metrics through the storage system's built-in logging mechanism: life time of each temporary and output file; the size of each temporary and output file; total amount of read and write I/O ops; average size of read and write I/O ops; and total bytes read and written.

Large-Scale Simulation Setup. We conduct extensive simulation using real production traces at the scale of a cluster. The simulations allow us to perform detailed study of performance and trade-offs at a large scale. Our simulation executes job placement on either SSD or HDD. If a job is placed on SSD but only partially fits, the remaining portion of the job spills over to HDD after filling the available SSD capacity. For experiments varying SSD capacities, we initially set the SSD constraint to infinity to determine the cluster's maximum space usage. We then simulate different scenarios by varying the SSD space quota.

Methods Compared. We compare 7 methods: FirstFit (simple heuristic), Heuristic (advanced heuristic, a modified [38]), ML Baseline (following [42]), adaptive hash (our method without ML models), adaptive ranking (our method), oracle TCIO (best bound when optimizing TCIO), and oracle TCO (best bound when optimizing TCO).

5.2 Integration in Real Systems [RQ1]

A range of pipelines are selected to generate a variety of I/O workloads in this prototype. These pipelines are compute programs that process large datasets, which involve GroupBy operations using a wide range of keys through multiple shuffle jobs. One category of pipelines is more cost-effective when using HDD, while the other category is more cost-effective to run on SSD. These pipelines are executed continuously in a production cloud cluster, akin to real production pipelines. We set up a dedicated SSD cache so that more precise and disturbance-free results can be measured. A total of 320 worker servers are used to execute the workloads, which includes 16 pipelines and 1024 shuffle jobs in total. The pipelines' combined peak storage usage is 3.6 TiB.

We pick one cluster and implemented two methods in *AnonCorp*'s storage system: FirstFit and Adaptive Ranking (ours). We vary the SSD quota to be 1.0% and 20% of the peak theoretical SSD usage limit (Figure 5). For the 1.0% case, our algorithm shows 1.14% TCO savings (**4.38**× over FirstFit). Our method gives 2.48% TCO savings (**1.77**× over FirstFit) in the 20% of the peak workload space usage. The TCIO savings indicate a similar pattern: Adaptive Ranking is 3.90× and 1.69× over FirstFit in the two SSD quota cases respectively.

The end-to-end prototype demonstrates the viability of our design. The measured savings of Adaptive Ranking and improvements over the baseline are in line with the performance in the large-scale simulation studies in Section 5.3 and thus validate our simulation methodology.

5.3 Overall Savings [RQ2, RQ3]

We pick 10 large TCO clusters to evaluate the overall savings. A typical workload space usage pattern is shown in Figure 6: We evaluate the peak SSD space usage by putting all the jobs on SSD, assuming unlimited SSD quota (dotted line). The solid lines (blue, orange, green, red) are for SSD space usage when we set the SSD quota to 100.0%, 50.0%, 10.0%, and 1.0%

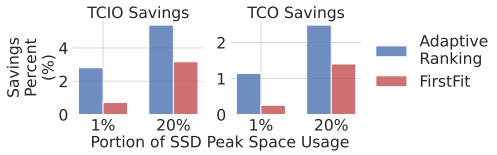

Figure 5. Prototype results.

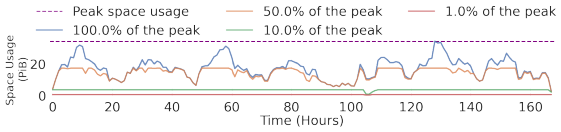

Figure 6. Space usage pattern.

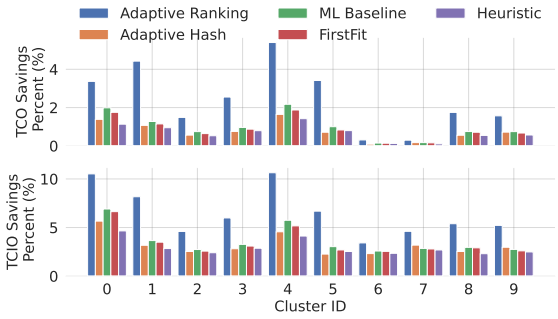

Figure 7. TCO savings (top) and TCIO savings (bottom) from different clusters with fixed SSD quota.

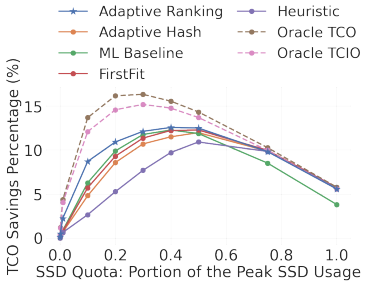

Figure 8. TCO savings.

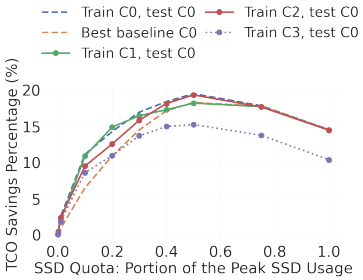

Figure 9. Cluster generalization.

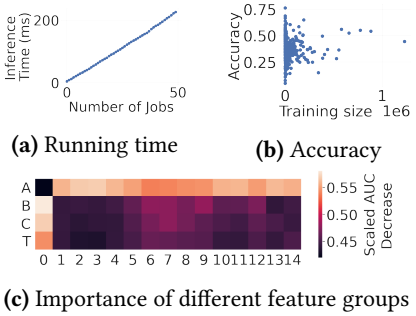

Figure 10. Model analysis.

of the peak space usage. This is a typical workload in cloud centers where many jobs run and the space usage is stable.

To show performance across different clusters, we fix the SSD quota at 1.0% of the peak SSD space usage and show savings per cluster (Figure 7). Our method (Adaptive Ranking) can save over 2.48× at maximum compared with baselines in terms of TCO savings. The TCIO savings follow a similar pattern. Traditionally, the TCIO savings increase as allowed SSD quota increases because SSD cost is not considered. In comparison, the TCO savings initially increase as SSD quota goes up but drop when SSD quota is very large due to high maintenance costs of SSD. We consider our approach as an effective solution especially when SSD space is limited.

We also evaluate the TCO savings when SSD quota varies because in practice, we want an approach that can adapt to different external factors (such as different SSD quota). Our method consistently saves more TCO than baselines, especially in limited SSD quota cases (Figure 8). The gap between our method (adaptive ranking) and adaptive hash (non-ML) clearly indicates the necessity of our category model. The gap between oracle (best possible in theory) and our method also indicates the improvement room for future works.

5.4 Generalizability [RQ4]

Another topic we explore is whether our method can generalize to new clusters, users, and pipelines. In practice, good

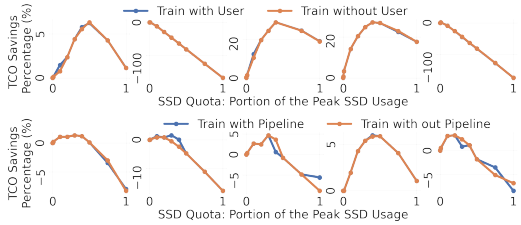

Figure 11. Generalization on new users (upper) and new pipelines (lower). Each figure is for one cluster.

generalizability is necessary as infrastructure, user behaviors, workloads, etc. change over time.

First we evaluate the generalizability across clusters in Figure 9. We train our category model on $C0, C1, C2, C3$, and evaluate their performance on $C0$, where $C3$ is a special cluster that only runs particular pipelines that are rare in other clusters. We find that our method can adapt to unseen clusters when the clusters run general pipelines.

Second, we evaluate our solution performance on new incoming users' jobs or jobs from unseen pipelines. We pick the second-largest TCO consuming user and pipeline, which are in different clusters. In evaluation, we compare two training methods: 1). We train the category model with the historical workloads including the user or the pipeline. 2). We train the category model with the historical workloads excluding

the user or the pipeline. We evaluate the TCO savings curve under five clusters and show that our method can achieve similar TCO savings on new users or pipelines as in the case where the users or pipelines are in the training set. We show the results in Figure 11 where the blue line (user or pipeline included in the training) and the orange line (user or pipeline excluded in the training) achieve similar savings online.

5.5 Model Analysis [RQ5]

Another practical requirement for ML models is their running latency and explainability. We show the accumulated inference time of 50 jobs in Figure 10a, where running the model takes about 4 ms per job, fast enough for making online placement decisions. Note that our ML model invocation is currently a prototype implemented in Python, without specific performance optimizations. Potential improvements in running efficiency could be achieved through further optimization, such as wrapping the ML models in C/C++, which is known for enhanced computational efficiency.

We show our model accuracy in Figure 10b. We compare the relationship between model accuracy and the training size across all the clusters. The average top-1 accuracy is 0.36 for 15-category classification model and we do not spot a strong correlation between training size and accuracy, which indicates that large data size may not be a requirement for applying ML in systems.

Next, we analyze how the model makes decisions by assessing feature importance across feature groups, as depicted in Figure 10c. Feature group significance is represented by color intensity, with lighter shades indicating higher importance. As discussed in Section 4.2, the features are divided into four groups: A (Historical system metrics), B (Execution metadata), C (Allocated resources), and T (Job timestamp).

To quantify the importance of features in predicting each category, we perform a binary prediction analysis (whether a job belongs to the category or not) for each category. For each feature, we measure the decrease in the area under the ROC curve (AUC) when that specific feature is excluded from binary prediction tasks. This approach helps us understand how the absence of a feature affects the model's predictive performance. These importance scores are normalized for comparability within each category. We calculate and present average importance scores for each feature group. In our model, the category 0 is for negative TCO savings and the remaining categories are associated with the ranking of I/O density. Our findings reveal that historical system metrics significantly influence the prediction of I/O density rankings. In contrast, the start time and execution metadata are more critical for predicting whether a job's TCO saving is negative.

We also find that our end-to-end savings may not benefit from more accurate models in Figure 12. Here, we evaluate the performance of the TCO savings assuming we have perfect models. The "Predicted category" is our approach. The "True category" is a method where we replace the category

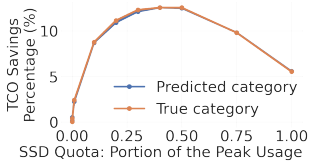

Figure 12. Comparison with using true category.

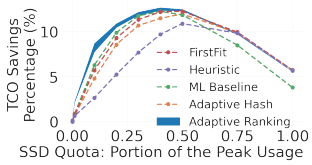

Figure 13. Adaptive algorithm parameters sensitivity.

Method	TCO Savings Percent	Model Top-1 Accuracy
Ours ($N = 2$)	9.25%	73.4%
Ours ($N = 5$)	11.1%	55.6%
Ours ($N = 15$)	12.7%	32.3%
Ours ($N = 25$)	12.6%	24.2%
Ours ($N = 35$)	10.8%	21.2%
Best Baseline	10.7%	/

Table 4. The TCO savings under different category numbers.

model's prediction results with the ground truth category. The latter is the case where we have 100% accuracy in our approach. We think this interesting observation can help us rethink the solutions of ML for systems. The challenges of ML for systems are not only about the accuracy and learning algorithm but also about how to formulate the learning problem and how to use the ML model.

5.6 Sensitivity Analysis [RQ6]

Adaptive Algorithm Parameters. We include all combinations of hyperparameters where $T_{\text{SpilloverTCIO}} \in \{[0.005, 0.03], [0.01, 0.15], [0.05, 0.25]\}$, look back window time length (seconds) $t_w \in \{600, 900, 1800\}$, and admission decision effective time $t_l \in \{600, 900, 1800\}$. We evaluate the sensitivity of the TCO savings for the same set of clusters in Figure 8. For each parameter combination, we apply the same parameter settings to all the clusters in the group. In Figure 13, the blue area in the figure presents the upper bound and lower bound of TCO savings under different SSD capacities across different hyperparameter combinations. Our solution is not sensitive in terms of hyperparameter selection in the adaptive algorithm.

Sensitivity on Category Numbers. Our evaluation utilizes the 0.1 SSD portion setting with all the algorithm parameters maintain consistent. It is critical to select an appropriately large number of categories to enable the model to effectively distinguish the cost across jobs without increasing the model's capacity for fine-grained category prediction. We present the impact of category numbers N on end-to-end TCO savings in Table 4. A model with smaller category number achieves higher accuracy but fails to optimize the end-to-end TCO savings due to its limited granularity. Conversely, increasing the number of categories enhances granularity but at the cost of accuracy, diminishing the TCO savings.

Figure 14. Category change of one cluster workload. From top to bottom, the SSD quota covers 0.01%, 1.0%, 10%, and 50% of the peak SSD space usage under no SSD quota limit. The green line is the observed SpilloverTCIO and the orange line represents the category admission threshold. The red area at the bottom of each figure is $T_{\text{SpilloverTCIO}}$.

5.7 Adaptive Category Selection Dynamics [RQ7]

To demonstrate the dynamics of our adaptive algorithm, we present the pattern of category threshold change and spill over percentage in Figure 14. We track the threshold change for 1 week online. Our adaptive category selection algorithm can adjust the category admission threshold to a higher range when SSD quota is limited and allow more category admissions when SSD space is plentiful.

6 Related Works

Machine Learning in Storage Systems. Prior works have shown the viability of machine learning for task property prediction in storage systems. [12] leverages a small neural network to infer SSD performance with fine granularity and help parallel storage applications. The method learns a binary latency model and pre-calculate an inflection point for each model during a labelling stage. The key benefit is model simplicity and fine granularity of prediction, enabling more complicated applications online within latency requirements. [42] tackles a problem related to our setting in data placement with methods that leverage application-level information and distributed traces in a way inspired by ideas from natural language processing. While the paper focuses on a specific learning problem of mapping textual metadata to storage-related properties, our work focuses on the practical designs and deployment of such models.

Other applications of machine learning in storage systems include training one monolithic model for the entire storage system (not deployable in warehouse-scale setting due to adaptability): applying imitation learning for cache replacement to approximate an optimal oracle policy [22], guiding the placement algorithm model through reinforcement learning [15, 32]; predicting properties in other aspects of data placement: improving a storage system through optimizing readahead and NFS read-size values with machine learning models [2], utilizing ML to improve on existing cache replacement strategies (LRU, LFU, etc.) [33], and predicting future task failures through ML [5].

Multiple machine learning techniques have also been proposed in broader system problems [16, 23], ranging from resource allocation [24], memory access prediction [13], offline storage configuration recommendation [19], database query optimization [20], to networking applications [1, 6]. Although the nature of these applications is different from data placement in storage systems, they all show evidence that machine learning can be used in systems and benefits from domain-specific formulations.

Data Placement in Practice. Though machine learning for systems has been widely explored in different application domains, the state of the art practical solutions for caching or tiering in storage systems are still mostly heuristic.

Hadoop offers three caching schedulers: FIFO [27], Capacity [29], Fair [39]. Spark supports FIFO, Fair. For each user, Azure tracks the last-accessed files and make the placement based of the self-tracked access history [7]. [38] presents a novel adaptive cache admission solutions for Google, of which we implement a modified version in our comparison.

Very recent works have also started rethinking the best practical solution within the heuristic-based domain. [36, 37] consider a modified FIFO for cache eviction, which achieves good scalability with high throughput on production traces from Twitter and MSR. [9] revisits the effectivenss of LRU versus FIFO and finds that FIFO exhibits better overall cost than LRU on production traces, including IBM COS traces. [41] proposes new heuristics for storage, specifically tailored for machine learning workloads at Meta.

Another noteworthy work presents a solver-based solution for task scheduling in the setting where each task contains a list of preferred locations identified prior to scheduling. Their approach formulates the problem as a minimum cost maximum matching problem [14]. Although closely related to our work, as discussed in the Section 2 and Section 3, the method is not directly feasible in our context. The primary challenge in adopting such a solver-based approach in our setting lies in the lack of jobs' cost at scheduling time.

7 Conclusion

We have presented a practical approach to data placement in data centers. In the process, we identify and solve practical challenges with a cross-layer data placement solution combining application-level ML models with storage-level heuristics. Our approach shows significant TCO savings over the state of the art techniques. We believe that our cross-layer approach presents a methodology for practical ML usage in systems beyond the data placement problem.

References

[1] Soheil Abbasloo, Chen-Yu Yen, and H Jonathan Chao. 2020. Classic meets modern: A pragmatic learning-based congestion control for the internet. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 632–647.

[2] Ibrahim Umit Akgun, Ali Selman Aydin, Andrew Burford, Michael McNeill, Michael Arkhangelskiy, and Erez Zadok. 2023. Improving Storage Systems Using Machine Learning. *ACM Transactions on Storage* 19, 1 (2023), 1–30.

[3] Christoph Albrecht, Arif Merchant, Murray Stokely, Muhammad Waliji, François Labelle, Nate Coehlo, Xudong Shi, and C Eric Schrock. 2013. Janus: Optimal flash provisioning for cloud storage workloads. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 91–102.

[4] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. 2011. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 143–157.

[5] Chandranil Chakraborttii and Heiner Litz. 2020. Improving the accuracy, adaptability, and interpretability of SSD failure prediction models. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 120–133.

[6] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. 2018. {PCC} Vivace:{Online-Learning} Congestion Control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 343–356.

[7] Ken Downie, Nataraj Sindam, Tamra Myers, Jeff Patterson, Will Gries, and Fabian Uhse. 2023. Cloud tiering overview: Cloud tiering heatmap. https://learn.microsoft.com/en-us/azure/storage/file-sync/file-sync-cloud-tiering-overview#cloud-tiering-heatmap.

[8] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–16.

[9] Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen Kat. 2020. It's Time to Revisit {LRU} vs.{FIFO}. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*.

[10] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 29–43.

[11] Mathieu Guillame-Bert, Sebastian Bruch, Richard Stotz, and Jan Pfeifer. 2023. Yggdrasil Decision Forests: A Fast and Extensible Decision Forests Library. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2023, Long Beach, CA, USA, August 6-10, 2023*. 4068–4077. https://doi.org/10.1145/3580305.3599933

[12] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S Gunawi. 2020. {LinnOS}: Predictability on Unpredictable Flash Storage with a Light Neural Network. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 173–190.

[13] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning memory access patterns. In *International Conference on Machine Learning*. PMLR, 1919–1928.

[14] Herodotos Herodotou and Elena Kakoulli. 2021. Trident: task scheduling over tiered storage systems in big data platforms. *Proceedings of the VLDB Endowment* 14, 9 (2021), 1570–1582.

[15] Ravi Kaler and Durga Toshniwal. 2023. Deep Reinforcement Learning based Data Placement optimization in Data Center Networks. In *2023 IEEE International Conference on Big Data (BigData)*. IEEE, 2293–2302.

[16] Marios Evangelos Kanakis, Ramin Khalili, and Lin Wang. 2022. Machine Learning for Computer Systems and Networking: A Survey. *Comput. Surveys* 55, 4 (2022), 1–36.

[17] Minguk Kang and Jaesik Park. 2020. Contragan: Contrastive learning for conditional image generation. *Advances in Neural Information Processing Systems* 33 (2020), 21357–21369.

[18] Hyojun Kim, Sangeetha Seshadri, Clement L Dickey, and Lawrence Chiu. 2014. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. *ACM Transactions on Storage (TOS)* 10, 4 (2014), 1–21.

[19] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2018. Selecta: Heterogeneous cloud storage configuration for data analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 759–773.

[20] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2021. Sagedb: A learned database system. (2021).

[21] Honglak Lee, Peter Pham, Yan Largman, and Andrew Ng. 2009. Unsupervised feature learning for audio classification using convolutional deep belief networks. *Advances in neural information processing systems* 22 (2009).

[22] Evan Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. 2020. An imitation learning approach for cache replacement. In *International Conference on Machine Learning*. PMLR, 6237–6247.

[23] Martin Maas. 2020. A taxonomy of ML for systems problems. *IEEE Micro* 40, 5 (2020), 8–16.

[24] Nikita Mishra, Connor Imes, John D Lafferty, and Henry Hoffmann. 2018. CALOREE: Learning control for predictable latency and low energy. *ACM SIGPLAN Notices* 53, 2 (2018), 184–198.

[25] Rani Molla. 2018. Google, Amazon and Microsoft cloud businesses helped more than double spending on data centers last year. https://www.vox.com/2018/3/15/17124300/google-amazon-microsoft-cloud-200-\percent-jump-data-center-acquisitions.

[26] Aizhamal Nurmamat kyzy, Aljoscha Krettek, Ahmet Altay, Ankur Goenka, Anton Kedin, Bruno Volpato, Charles Chen, Chad Dombrova, Chamikara Jayalath, Danny McCormick, David Cavazos, Davor Bonaci, Dan Halperin, Emily Ye, Frances Perry, Harshit Dwivedi, Heejong Lee, Henry Suryawirawan, Ismaël Mejía, James Malone, Jesse Anderson, John Casey, Julien Phalip, Jack R. McCluskey, Kiley Sok, Kenneth Knowles, Leonid Kuligin, Mark Liu, Mikhail Gryzykhin, Robert Bradshaw, Tyler Akidau, Thomas Groh, Thomas Weise, Eugene Kirpichov, Jean-Baptiste Onofré, Anand Iyer, Alexey Romanenko, Pablo Estrada, Rafael Fernández, Matthias Baetens, Reza Rokni, Tanay Tummalapalli, Udi Meiri, Boyuan Zhang, Rui Wang, Maximilian Michels, Ning Kang, Pedro Galvan, Rion Williams, Saavan Nanavati, Brian Hulette, Robert Burke, Valentyn Tymofieiev, Andrew Pilloud, Kyle Weaver, Daniel Oliviera, Robin Qiu, Mark Zeng, Yifan Zou, Artur Khanin, Ilya Kozyrev, Alex Kosolapov, Brittany Hermann, Svetak Sundhar, Israel Herraiz, Yichi Zhang, Danielle Syse, Ritesh Ghorse, Yi Hu, Pablo Rodriguez Defino, Namita Sharma, and Talat Uyarer. 2012. Apache Beam: An advanced unified programming model. https://beam.apache.org.

[27] Seyed Reza Pakize. 2014. A comprehensive view of Hadoop MapReduce scheduling algorithms. *International Journal of Computer Networks & Communications Security* 2, 9 (2014), 308–317.

[28] Andrei Paleyes, Raoul-Gabriel Urma, and Neil D Lawrence. 2022. Challenges in deploying machine learning: a survey of case studies. *Comput. Surveys* 55, 6 (2022), 1–29.

[29] Aparna Raj, Kamaldeep Kaur, Uddipan Dutta, V Venkat Sandeep, and Shrisha Rao. 2012. Enhancement of hadoop clusters with virtualization using the capacity scheduler. In *2012 Third International Conference on Services in Emerging Markets*. IEEE, 50–57.

[30] Mohit Saxena, Michael M Swift, and Yiying Zhang. 2012. Flashtier: a lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM european conference on Computer Systems*. 267–280.

[31] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 1–10. https://doi.org/10.1109/MSST.2010.5496972

[32] Gagandeep Singh, Rakesh Nadig, Jisung Park, Rahul Bera, Nastaran Hajinazar, David Novo, Juan Gómez-Luna, Sander Stuijk, Henk Corporaal, and Onur Mutlu. 2022. Sibyl: Adaptive and extensible data placement in hybrid storage systems using online reinforcement learning. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 320–336.

[33] Giuseppe Vietri, Liana V Rodriguez, Wendy A Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. 2018. Driving cache replacement with {ML-based} {LeCaR}. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*.

[34] Reynold Xin. 2014. Apache Spark the Fastest Open Source Engine for Sorting a Petabyte. https://www.databricks.com/blog/2014/10/10/spark-petabyte-sort.html. Accessed: 2024-01-22.

[35] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, Swaminathan Sundararaman, and Robert Wood. 2013. HEC: improving endurance of high performance flash-based cache devices. In *Proceedings of the 6th International Systems and Storage Conference*. 1–11.

[36] Juncheng Yang, Ziyue Qiu, Yazhuo Zhang, Yao Yue, and KV Rashmi. 2023. FIFO can be Better than LRU: the Power of Lazy Promotion and Quick Demotion. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*. 70–79.

[37] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. 2023. FIFO queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 130–149.

[38] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister. 2022. {CacheSack}: Admission Optimization for Google Datacenter Flash Caches. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 1021–1036.

[39] Matei Zaharia, Dhruba Borthakur, J Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2009. *Job scheduling for multi-user mapreduce clusters*. Technical Report. Technical Report UCB/EECS-2009-55, EECS Department, University of California ….

[40] 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.

[41] Mark Zhao, Satadru Pan, Niket Agarwal, Zhaoduo Wen, David Xu, Anand Natarajan, Pavan Kumar, Ritesh Tijoriwala, Karan Asher, Hao Wu, et al. 2023. {Tectonic-Shift}: A Composite Storage Fabric for {Large-Scale} {ML} Training. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 433–449.

[42] Giulio Zhou and Martin Maas. 2021. Learning on distributed traces for data center storage systems. *Proceedings of Machine Learning and Systems* 3 (2021), 350–364.