# MAST: Global Scheduling of ML Training across Geo-Distributed Datacenters at Hyperscale

Arnab Choudhury, *Meta Platforms;* Yang Wang, *Meta Platforms and The Ohio State University;* Tuomas Pelkonen, *Meta Platforms;* Kutta Srinivasan, *LinkedIn;* Abha Jain, Shenghao Lin, Delia David, Siavash Soleimanifard, Michael Chen, Abhishek Yadav, Ritesh Tijoriwala, Denis Samoylov, and Chunqiang Tang, *Meta Platforms*

https://www.usenix.org/conference/osdi24/presentation/choudhury

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

# MAST: Global Scheduling of ML Training across Geo-Distributed Datacenters at Hyperscale

Arnab Choudhury[1], Yang Wang[1 †], Tuomas Pelkonen[1], Kutta Srinivasan[‡], Abha Jain[1],
Shenghao Lin[1], Delia David[1], Siavash Soleimanifard[1], Michael Chen[1], Abhishek Yadav[1],
Ritesh Tijoriwala[1], Denis Samoylov[1], and Chunqiang Tang[1]

[1] Meta Platforms
[†] The Ohio State University
[‡] LinkedIn (work done while at Meta)

## Abstract

In public clouds, users must manually select a datacenter region to upload their ML training data and launch ML training workloads in the same region to ensure data and computation colocation. Unfortunately, isolated decisions by individual users can lead to a mismatch between workload demand and hardware supply across regions, hurting the cloud provider's hardware utilization and profitability. To address this problem in Meta's hyperscale private cloud, we provide a global-scheduling abstraction to all ML training workloads. Users simply submit their training workloads to MAST, our global scheduler, and rely on it to intelligently place both data and training workloads to different regions. We describe three design principles that enable MAST to schedule complex ML training workloads at a global scale: temporal decoupling, scope decoupling, and exhaustive search. MAST successfully balances the load across global regions. Before MAST, the most overloaded region had a GPU demand-to-supply ratio of 2.63 for high-priority workloads. With MAST, this ratio has been reduced to 0.98, effectively eliminating the overload.

## 1 Introduction

The success of ML applications [8, 46] has resulted in ML training becoming the fastest-growing datacenter workload. Public cloud providers run ML training workloads in multiple geo-distributed datacenter regions [3,4,15] to ensure sufficient capacity. Accordingly, users need to manually select a region to upload their ML training data and then launch training workloads in the same region to ensure colocation of data and computation. Unfortunately, such manual selection can lead to a regional mismatch between workload demand and hardware supply. For instance, one region may exhaust its capacity, accumulating a long queue of pending jobs, while another region has surplus capacity remaining idle.

Meta's private cloud used to experience this load imbalance. It comprised tens of datacenter regions, millions of machines, and tens of thousands of GPUs. Similar to public clouds, users initially had to manually select regions to store training data and launch workloads. Users' suboptimal decisions previously led to an imbalance in the GPU demand-to-supply ratio, reaching as high as 2.63 in certain regions for high-priority workloads, which was later reduced to 0.98 through optimizations described in this paper.

While much research has been conducted on scheduling ML workloads in a single cluster [1, 2, 5–7, 9, 13, 17, 21, 23–25, 30, 31, 33–35, 39, 40, 45, 49, 51–54, 57, 59], there has been little effort to address the issue of regional mismatch between workload demand and hardware supply. To address this challenge, our private cloud has introduced the *global-scheduling* abstraction that shields users from the complexity of regions. With the global-scheduling abstraction, users simply submit their ML training workloads to our global scheduler called MAST (short for ML Application Scheduler on Twine [44]) and rely on it to intelligently place both training data and workloads into different regions.

To provide the global-scheduling abstraction, MAST faces two major challenges:

- *Data-GPU colocation*: Without careful coordination, there is a risk of location mismatch between GPUs and data. For instance, one region may have the necessary training data but run out of available GPUs, while another region may have available GPUs but lack the required training data. Due to the massive volume of training data and the limited cross-region network bandwidth, on-demand cross-region data migration can be both costly and time-consuming.
- *Scalability*: MAST allocates not only GPU machines for training but also CPU machines for data preprocessing [58]. As CPU machines may be dynamically reassigned across ML and non-ML workloads based on demand, conceptually, MAST needs to find machines to run ML workloads out of millions of machines spread across tens of regions. Global resource allocation at this scale has not been studied before.

Contributions: Chunqiang, Kutta, and Tuomas initiated the MAST project in 2020. In terms of paper writing, Yang wrote the majority of the paper, followed by Chunqiang. In terms of coding, Arnab, Kutta, and Tuomas led the project's development at different times. All other authors also made major contributions to the project's development.

We leverage three principles to address these challenges: *temporal decoupling*, *scope decoupling*, and *exhaustive search*. We elaborate on these principles below.

**Temporal decoupling.** We divide the scheduling responsibilities into two paths: a fast path for real-time job scheduling and a slow path that continually optimizes data and machine assignment in the background. The slow path intelligently replicates ML training data across regions, enabling the fast path to more easily colocate computation with data. Despite the relaxed timing, cross-region data placement remains very challenging. It requires continuous optimization of the placement of billions of data partitions across tens of geo-distributed regions, considering per-region capacity constraints and the data access pattern of millions of daily ML training jobs and analytics jobs from Spark [55] and Presto [42].

We model data placement as a mixed integer programming (MIP) problem, and the scarcity of GPUs drives novel decisions in our solution. Due to the high cost and demand of GPUs, we target maximizing GPU utilization. Imposing a hard constraint in the MIP problem that GPU demand must be lower than GPU supply in every region, as in prior work [20] for CPU and storage, often renders the problem unsolvable. Instead, MAST allows GPU oversubscription and preempts low-priority jobs as needed. This approach mandates a reassessment of the objective function and constraints in the MIP problem, not only for GPU-related terms but also for other resources that GPUs depend on. We share insights gained from multiple iterations refining the MIP problem through production experience (§3).

To tackle the scalability challenge, as illustrated in Figure 1, MAST adopts a three-level scheduling hierarchy: Global ML Scheduler (GMS)→Regional ML Scheduler (RMS)→Cluster Manager (CM). In addition to managing data placement, the slow path also helps scale RMS by constraining its search for available machines. It dynamically pre-assigns machines to *dynamic clusters*, allowing RMS to only search through machines within the ML dynamic clusters and disregard non-ML dynamic clusters.

**Scope decoupling.** A job scheduling system has three main responsibilities. First, it manages the job queue, which entails queuing and prioritizing jobs when there are insufficient resources to run all jobs. Second, it handles resource allocation, which involves computing bin-packing-like solutions by modeling machines as bins and tasks as objects. Third, it manages container orchestration, which executes the bin-packing plan, runs containers, and monitors their health. Traditional systems [19, 47, 48, 56] handle all these responsibilities within the same scope, i.e., within a cluster.

Our key insight is that sharing the same scope for all three responsibilities unnecessarily limits scalability, reducing the potentially larger scopes of job queue management and resource allocation to the minimal scope of container orchestration. Note that container orchestration is the least scalable due to its heavy duties and, consequently, has the smallest scope.

In contrast, as shown in Figure 1, our *scope-decoupling* principle allows the three responsibilities to operate at different scopes: (1) the job queue is managed by GMS at the global scope, covering all pending jobs for all regions; (2) resource allocation is managed by RMS at the regional scope, taking into account all machines in a region's ML dynamic clusters; and (3) container orchestration is managed by the CM at the smallest dynamic-cluster scope. This approach allows job queue management and resource allocation to operate at bigger scopes to minimize stranded resources and optimize job placement. A key challenge is to make GMS and RMS sufficiently scalable to operate at their bigger scopes, which is further discussed in §4.2.1 and §4.3.1.

**Exhaustive search.** Existing systems [10, 20, 28] often adopt the federation approach to scale out. When a new job arrives, the Federation Manager employs simple heuristics to assign the job to the least loaded cluster, and then its cluster manager manages all subsequent operations, including job queuing, resource allocation, and container orchestration. However, as ML training clusters are almost always fully utilized, scheduling a new job often requires a complex decision to preempting existing lower-priority jobs. This complexity makes the simplistic federation approach less effective.

Our key insight is that, unlike short-lived analytics jobs [10, 38, 41, 50], ML training jobs often run for extended periods on expensive GPUs. Therefore, instead of searching just one cluster to allocate resources hastily, it is beneficial to conduct an *exhaustive search* across all relevant clusters for higher quality placement. As depicted in Figure 1, MAST's multiple RMSs can concurrently compute resource allocation plans for one job in different regions, with the optimal plan determined through a final auction process. A key hurdle is ensuring the scalability of RMS, which is discussed in §4.3.1.

**Contributions.** We make the following contributions.

- We propose the *global-scheduling* abstraction to shield users from the complexity of geo-distributed datacenters and improve hardware utilization through joint placement of data and training workloads across regions.
- We propose three principles–*temporal decoupling*, *scope decoupling*, and *exhaustive search*–to achieve high-quality data and computation placement in a scalable manner.
- We demonstrate the effectiveness of global ML scheduling through our hyperscale deployment of MAST and validate its design using production data.

## 2 Background of ML Training at Meta

In this section, we provide the necessary background to set the stage for future discussions.

**Datacenter and hardware.** Our private cloud comprises tens of regions and millions of machines. A region comprises

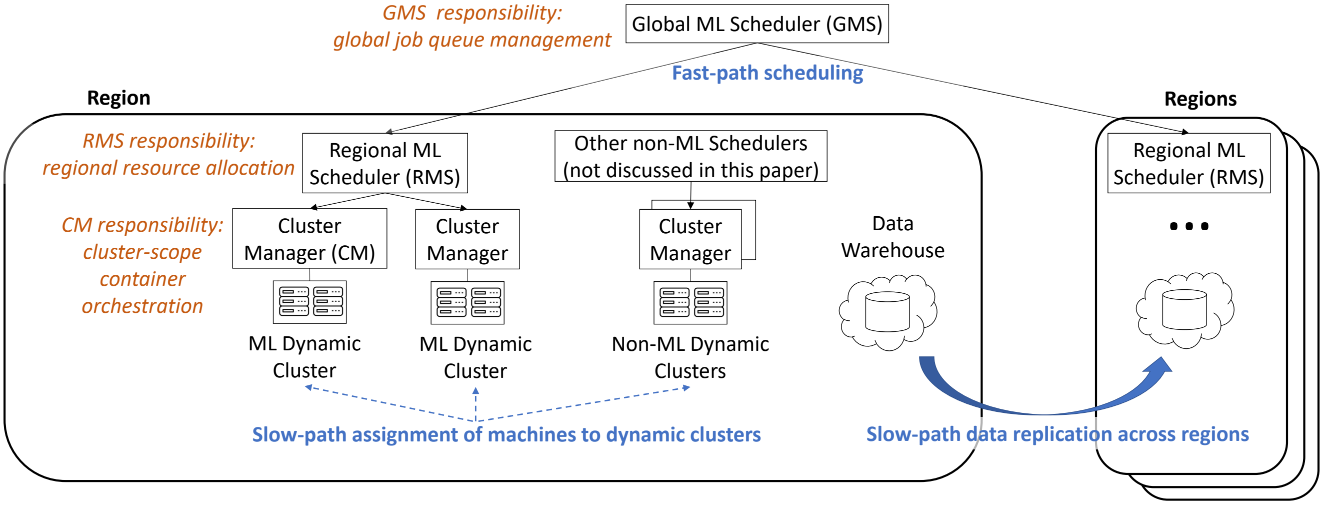

GMS responsibility:
global job queue management
Global ML Scheduler (GMS)
Fast-path scheduling
Region
Regions
RMS responsibility:
regional resource allocation
Regional ML Scheduler (RMS)
Other non-ML Schedulers (not discussed in this paper)
CM responsibility:
cluster-scope container orchestration
Cluster Manager (CM)
Cluster Manager
Cluster Manager
Data Warehouse
Regional ML Scheduler (RMS)
ML Dynamic Cluster
ML Dynamic Cluster
Non-ML Dynamic Clusters
Slow-path assignment of machines to dynamic clusters
Slow-path data replication across regions

Figure 1: Conceptual architecture of MAST. Global ML Scheduler (GMS), Regional ML Schedule (RMS), and Cluster Manager (CM) handle different scheduling responsibilities at different scopes: global, regional, and cluster, respectively.

multiple datacenters that are close to one aother. The cross-region network bandwidth is about 10 times lower than the bisection bandwidth between datacenters within a region. Parts of a datacenter are occupied by ML training clusters, with machines equipped with multiple GPUs and connected by both 8x200Gbps RoCE network and 4x100Gbps Ethernet.

ML training is data intensive and prefers colocating the compute and data of a training workload. For tasks that belong to an ML training workload, we prefer to place them in the same rack, cluster, datacenter, and region, in that order. Separating the compute and data across regions or placing tasks in different regions would result in unacceptable performance.

Historically, datacenter hardware has been procured incrementally depending on the specific needs at different times, resulting in uneven distribution of hardware types across regions. This is discussed in Flux [11] and also shown in Figure 2. This disparity makes colocation of data and compute difficult, requiring global optimization. For example, since *Region6* is short of GPUs, it is better to place data used by CPU-based analytics jobs in *Region6*. If a few GPU-based ML training workloads share the same data as those analytics jobs, we should schedule them in *Region6* as well. However, if there are too many such ML workloads, we will have to replicate their data to other regions and execute them there.

**Dynamic clusters.** As shown in Figure 1, a slow-path component called RAS pre-assigns machines to *dynamic clusters*, which are known as "*reservations*" in the RAS paper [36]. This enables the Regional ML Scheduler (RMS) to scale by searching only through machines that are within the ML dynamic clusters. Typically, an ML dynamic cluster comprises both GPU and CPU machines. To update dynamic clusters, RAS takes as inputs all machines in a region and the new or updated specification for each dynamic cluster's intended size and preference for certain hardware types. RAS formulates a MIP problem to allocate machines to dynamic clusters. MAST consumes the outputs of RAS (i.e., the dynamic clusters created by RAS), and MAST's scheduling decisions do not influence or feed back into RAS.

We provide a brief summary of RAS and refer readers to the RAS paper [36] for details. RAS ensures that the total machine capacity allocated to a dynamic cluster meets the requirements specified by administrators and includes sufficient buffers to handle both random and correlated machine failures. Correlated failures, such as power outages in large fault domains within a datacenter, can render tens of thousands of machines unavailable. RAS distributes a dynamic cluster's machines across different fault domains to ensure that sufficient healthy machines remain available when a large fault domain fails. Additionally, RAS reduces unnecessary cross-datacenter communication by ensuring a proper ratio of compute machines to storage machines in each datacenter. Finally, RAS reruns its optimization periodically (e.g., every 30 minutes) to adapt to changes. For example, when new datacenters are brought online, RAS can reduce the buffer size needed for handling correlated failures by further spreading out a dynamic cluster's machines into these new datacenters.

**ML training workload.** A training workload comprises multiple heterogeneous jobs, each job comprises multiple homogeneous tasks, and a task is mapped to a Linux container. Therefore, the hierarchy is workload→job→task. For example, a training workload may comprise (1) a training job that executes back-propagation training; (2) a data-preprocessing job [58]; (3) a parameter-server job; and (4) an evaluator job that evaluates the generated model. A workload's all tasks need gang scheduling, i.e., they must be allocated together. If a training job uses less than a full GPU, in theory, the GPU can be shared by multiple jobs using Multi-Instance GPU (MIG) [37] or other software approaches. In practice,

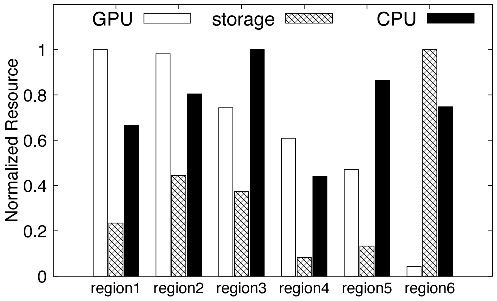


Figure 2: Uneven distribution of hardware across regions. Storage is normalized by capacity, and GPU and CPU are normalized by server count.

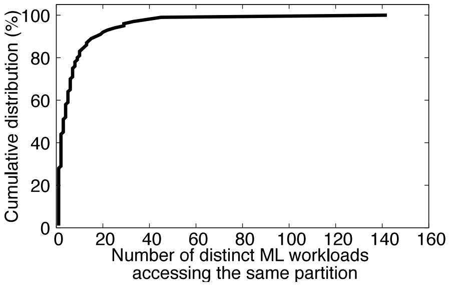


Figure 3: Hotness of data partitions, measured by the number of distinct ML workloads accessing each partition.

however, all our training jobs use at least one full GPU due to the large amount of training data.

**Data warehouse.** Our data warehouse stores exabytes of data in a three-level hierarchy: hundreds of namespaces→millions of tables→ billions of data partitions. A partition is immutable once created, but new partitions can be added to an existing table. For example, every day, the "*user_activity*" table can add a new partition to record user activities in the past 24 hours. Some data partitions are simultaneously used by ML training and data analytics, such as Spark [55] and Presto [42]. We have developed a system called Tetris, which optimizes data placement across regions, taking into account the data-access patterns of Spark, Presto, and ML training jobs.

**Sharing of data partitions by workloads.** Figure 3 shows that data partitions are often shared by multiple ML workloads. At the P50, P90, and P99 percentiles, a data partition is shared by 3, 17, and 45 distinct workloads, respectively. Data sharing complicates the problem of data placement, as migrating one data partition across regions may require the migration of multiple workloads dependent on the partition. Furthermore, it is necessary to replicate the hottest partitions across multiple regions to prevent load imbalance, as a large number of workloads dependent on those partitions will otherwise be forced to run in a small number of regions.

**Long execution time of ML training jobs.** ML training is resource intensive and can take a long time to finish. At Meta, ML training workloads often take 10 times longer to finish than Spark [55] analytics jobs. Therefore, a suboptimal placement decision has a bigger negative impact on ML training. This motivates the *exhaustive search principle* described in §1. Moreover, when workloads run longer on a larger number of machines, the workload scheduling throughput decreases. Therefore, as shown in Figure 1, it is feasible to manage the job queue and resource allocation at the global and regional scope, respectively, rather than at the smaller cluster scope that leads to more fragmentation.

**Quota and job preemption.** Training workloads with different priorities are assigned capacity quotas per priority level. If a team's capacity usage is within their quota, MAST guarantees starting their training workloads within a certain latency. Once a team exceeds their quota, they can still submit workloads to run opportunistically at the lowest priority, subject to preemption when a higher-priority workload arrives. Consequently, training clusters are always fully utilized due to low-priority workloads for experimental purposes. Scheduling a new workload often involves a complex decision to preempt lower-priority jobs. This complexity renders a simple Federation Manager less effective.

**Checkpoint for recovery.** A training workload periodically checkpoints its state. When a machine fails, the cluster manager restarts its workload on a replacement machine, allowing it to recover its state from the checkpoint and resume execution. Before preempting a low-priority workload for a high-priority one, it also saves a checkpoint for later restoration. As we continuously reduce the time needed to save a checkpoint, we are moving towards more frequent checkpoints to minimize the amount of lost work between two checkpoints during recovery. This has become increasingly important as the size of training workloads for large-language models keeps growing, and recovery becomes more costly.

**Separate application-level schedulers for ML and non-ML workloads.** As depicted in Figure 1, ML and non-ML workloads are managed by distinct schedulers. The extensible architecture of Twine [44] allows all workloads to share a common cluster manager for machine and container management, while employing different application-level schedulers for specific workloads. For instance, MAST is used for ML training workloads, Shard Manager [29] for stateful databases, Turbine [32] for stream processing, and Chronos for analytics jobs. Each of these application-level schedulers is optimized for a specific purpose. Shard Manager, for example, is optimized for high database availability, Chronos for high scheduling throughput of short-lived analytics jobs, and MAST for high-quality decisions and data-GPU colocation.

## 3 Slow-path Data Placement

To enable global ML scheduling, it is crucial to have both the necessary hardware and training data for an ML workload available in certain datacenter regions simultaneously. Following the *temporal-decoupling* principle, MAST optimizes cross-region data placement on a daily basis using a slow-path component called Tetris, which determines data placement and replication for the underlying storage system. Given that data analytics (e.g., Spark [55] and Presto [42]) and ML training can concurrently access the same data, Tetris jointly optimizes data placement for them. The entities accessing the data, such as Presto queries, Spark jobs, and ML workloads, are collectively referred to as "*jobs*" for simplicity.

## 3.1 Context of Data Placement

Recall that our data warehouse uses a three-level hierarchy: hundreds of namespaces→millions of tables→billions of data partitions. The large number of data partitions presents a significant scalability challenge for data placement. To improve scalability, Tetris determines data placement in two steps: first by placing tables to regions, which means that all partitions of a table must reside in the same region; and then by placing partitions to data centers within each region. Since the algorithms for both steps are similar, we mainly present the table-to-region placement algorithm.

Each table has a home datacenter region, which is where its new data is generated. Subsequently, the table may be replicated to other regions for various purposes. If an ML job takes a table as input, then at least one of the table's regions should have the types of GPUs required by the job. We refer to this property as "*data-GPU collocation*." Due to the large size of our data warehouse (exabytes) and limited cross-region network bandwidth, replicating data takes time. Consequently, a table's replicas in other regions are more stale compared to those in the home region. Therefore, high-priority jobs requiring fresh data must run in the home regions of their input tables, and these home regions must have the types of GPUs required by those jobs. We call this property "*training-at-home-region*." For low-priority jobs, *training-at-home-region* is preferred but not required.

Tetris first determines the home region of each table and then determines replica regions. We are still evaluating the feasibility of determining both simultaneously. While it may result in improved placement, currently we find its computational cost to be too high.

To plan data placement, Tetris requires the usage information of each job's input tables, hardware needs, and estimated runtime. High-priority training jobs, with trained models deployed for immediate production use, are typically retrained daily or more frequently with updated data. The usage information of such recurring jobs rarely changes, so Tetris can derive this information from their historical data. Tetris does not predict usage information for new training jobs that show up for the first time. Instead, MAST's fast-path online scheduling manages first-time jobs upon submission. If no region has both the necessary data and hardware to run a first-time job, MAST will initiate data movement and wait for its completion before scheduling the job. However, many first-time jobs benefit directly from data placement plans for recurring jobs and will not be blocked on data replication, as jobs from the same team frequently share input tables and require the same types of hardware. For instance, multiple one-time experimental jobs are often submitted to fine-tune a parameter of a production recurring job. Our production data in §5 shows that although about 70% of the jobs are first-time jobs, only a small fraction of them trigger on-demand data movement. Note that on-demand data movement may also be triggered due to failures or in rare cases where a recurring job changes its input tables or hardware needs.

## 3.2 Problem Formulation Overview

We formulate home-region placement as the mixed-integer programming (MIP) problem shown in Figure 4. While MIP has been applied in resource allocation [11, 20], the key insights often lie in the details of a specific problem formulation. In Tetris, we need to carefully evaluate various approaches for resource allocations:

- The *hard-quota* approach mandates that resource demand must stay below supply in every region.
- The *hard-balance* approach does not enforce *hard-quota* but mandates that the overload situation (i.e., demand above supply) must not deteriorate for any region due to a new placement.
- The *soft-balance* approach does not enforce *hard-quota* or *hard-balance* but instead aims to balance the demand-to-supply ratio across all regions as much as possible.

After iteratively improving Tetris based on production experiences, we have learned that different resource types require different approaches. For GPUs, imposing hard constraints in the MIP problem, as in previous work for CPU and storage [20], often renders the problem unsolvable due to the scarcity of GPUs. Hence, we adopt the *soft-balance* approach and preempt low-priority jobs to accommodate high-priority jobs as needed. Specifically, we introduce a penalty if a region's GPU demand deviates from the ideal case where all regions' GPU demand-to-supply ratios are the same. In our initial implementation, the situations of overload and underload were penalized equally. However, in practice, overload is more problematic as it causes longer wait times for the impacted jobs. Therefore, our current implementation more severely penalizes overload.

For storage space, the *hard-quota* approach is necessary because we cannot delete data when demand exceeds supply. However, *hard-quota* alone is insufficient, as an imbalanced data distribution across regions may occur, leading to GPUs in some regions being bottlenecked on I/O bandwidth to access the data. Therefore, we also apply *soft-balance* to storage. In the past, we also experimented with using *hard-balance*, but it proved ineffective as it completely prevents moving data from a region to another region with a higher demand-to-supply rate, which is sometimes necessary for other goals.

Concretely, our MIP problem formulation aims to achieve the following soft goals:

1. Minimize cross-region traffic for reading data during training (Expression 1).
2. Balances the demand and supply of GPUs across regions (Expression 2).
3. Balances the demand and supply of storage space across regions (Expression 3).

In addition, it aims to meet the following hard constraints:

4. Each region has sufficient storage space for the tables it stores (Expression 4).
5. Each application's CPU usage does not exceed its quota (Expression 5). Here, applications refer to those generating data for or sharing data with ML training, such as analytics jobs. An application may comprise multiple jobs.
6. The demand of jobs satisfying the training-at-home-region property remains above a certain threshold (Expression 6).

### 3.3 Problem Formulation Details

This section can be safely skipped during the initial reading, as it mainly details the problem formulation.

**Minimize:**

$$w_1 \sum_{\text{job}_j} \sum_{\text{table}_i \in \text{job}_j} \text{size}(\text{table}_i) \;\; [1 - I(R(\text{job}_j), R(\text{table}_i))] \tag{1}$$

$$+ w_2 \sum_{\text{region}_i} \sum_{\text{GPU}_j} \sum_{P_k} w_{p_k} \sigma_1(\text{Demand}^{\text{region}_i}_{\text{GPU}_j, P_k}, \text{Supply}^{\text{region}_i}_{\text{GPU}_j}) \tag{2}$$

$$+ w_3 \sum_{\text{region}_i} \sigma_2(\text{Demand}^{\text{region}_i}_{\text{storage}}, \text{Supply}^{\text{region}_i}_{\text{storage}}) \tag{3}$$

**Subject to:**

$\forall\text{region} \;\; r$

$$\sum_{\text{table}_i} \text{size}(\text{table}_i) \;\; I(R(\text{table}_i), \text{region}_r) < \text{storage\_capacity}(r) \tag{4}$$

$$\forall\text{app} \sum_{\text{job}_j \in \text{app}} \text{CPU}(\text{job}_j) \;\; I(R(\text{job}_j), \text{region}_r) < \text{CPU\_capacity}(\text{app}, r) \tag{5}$$

$$\sum_{\text{high-priority job}_j} \text{GPU}(\text{job}_j) I(\text{GPU-Type}_{\text{job}_j}, R(\text{job}_j)) \geq \text{threshold} \tag{6}$$

**Where:**

$$\sigma_1(\text{Demand}^{\text{region}_i}_{\text{GPU}_j, P_k}, \text{Supply}^{\text{region}_i}_{\text{GPU}_j}) =$$

$$\left( \frac{\text{Demand}^{\text{region}_i}_{\text{GPU}_j, P_k}}{\sum_{\text{region}_i} \text{Demand}^{\text{region}_i}_{\text{GPU}_j, P_k}} - \frac{\text{Supply}^{\text{region}_i}_{\text{GPU}_j}}{\sum_{\text{region}_i} \text{Supply}^{\text{region}_i}_{\text{GPU}_j}} \right)^2 \tag{7}$$

$$\times \; \text{sigmoid}(\max(0, \text{Demand}^{\text{region}_i}_{\text{GPU}_j} - \text{Supply}^{\text{region}_i}_{\text{GPU}_j}) \tag{8}$$

$$\sigma_2(\text{Demand}^{\text{region}_i}_{\text{storage}}, \text{Supply}^{\text{region}_i}_{\text{storage}}) =$$

$$\left( \frac{\text{Demand}^{\text{region}_i}_{\text{storage}}}{\sum_{\text{region}_i} \text{Demand}^{\text{region}_i}_{\text{storage}}} - \frac{\text{Supply}^{\text{region}_i}_{\text{storage}}}{\sum_{\text{region}_i} \text{Supply}^{\text{region}_i}_{\text{storage}}} \right)^2 \tag{9}$$

Figure 4: Formulation of the data placement problem.

Among the expressions in Figure 4, $R(\text{table}_i)$ is the only decision variable, which determines the home region of $\text{table}_i$. The region in which $\text{job}_j$ is placed is represented by $R(\text{job}_j)$, which is not a decision variable and is inferred from the placement of tables. A job may access multiple tables. For Spark or Presto jobs, the home region of the majority of the tables accessed by the job determines $R(\text{job}_j)$. For an ML training $job_j$, if the home regions of all tables accessed by the job are the same, then their home region determines $R(\text{job}_j)$. Otherwise, Tetris sets $R(\text{job}_j)$ to NULL temporarily, and will fix it at a later stage by replicating tables to ensure that at least one region has all these tables (§3.5). Note that while a Presto or Spark job can read some of its input tables across regions, an ML job must read all its input tables from the local region. This difference is due to the fact that GPUs used by ML jobs are much more costly and should not be stalled on reading data during execution. Further note that $R(\text{job}_j)$ is an auxiliary variable used when determining $R(\text{table}_i)$. After the completion of table placement on the slow path, MAST's real-time job scheduling on the fast path (§4) has the freedom to place the job in a region different from $R(\text{job}_j)$, depending on the available resources and table replicas at the scheduling time. Similarly, all the *Demand* variables are also auxiliary variables, inferred from $R(\text{table}_i)$ and $R(\text{job}_j)$.

Other symbols are defined as follows: $\text{size}(\text{table}_i)$ (size of a table), $\text{CPU}(\text{job}_j)$ (CPU hours required by a job), $\text{GPU}(\text{job}_j)$ (GPU hours required by a job), and $\text{GPU-Type}_{\text{job}_j}$ (types of GPUs required by a job) are estimated from the past history of the tables and jobs. $\text{storage\_capacity}(r)$, $\text{CPU\_capacity}(\text{app}, r)$, and *Supply*, are set by the administrator. The weights, $w_1$, $w_2$, $w_3$, $w_{p_k}$, and $\text{w}_{\text{job}_j}$, can be tuned by the administrator to prioritize certain terms or jobs.

$I(a,b)$ is a binary operator that evaluates to 1 if its two operands meet certain conditions and 0 otherwise. Specifically, $I(R(\text{job}_j), R(\text{table}_i))$ checks if the job and the table are in the same region. $I(R(\text{job}_j), \text{region}_r)$ checks if the job is placed in region $r$. $I(R(\text{table}_i), \text{region}_r)$ checks if the table is placed in region $r$. $I(\text{GPU-Type}_{\text{job}_j}, R(\text{job}_j))$ checks if $\text{job}_j$'s region has the needed type of GPUs to run the job.

Expression 1 aims to minimize the total size of tables that are read across regions, i.e., when $\text{job}_j$ needs to access $\text{table}_i$ but they are not in the same region. The notation $\text{table}_i \in \text{job}_j$ denotes the tables accessed by $\text{job}_j$. For an ML training $\text{job}_j$, if its home region $R(\text{job}_j)$ is NULL (i.e., the home regions of the tables accessed by the job are not the same), Expression 1 will assume that all of these tables are accessed across regions.

Expression 2 *soft-balances* the supply and demand of GPUs across regions. $\text{GPU}_j$ is a specific type of GPU and $P_k$ is a job priority level. $\text{Demand}^{\text{region}_i}_{\text{GPU}_j, P_k}$ is the demand for $\text{GPU}_j$ at $\text{region}_i$ at priority level $P_k$. $\text{Supply}^{\text{region}_i}_{\text{GPU}_j}$ is the supply for $\text{GPU}_j$ at $\text{region}_i$. Demands and supplies are measured in GPU hours. Expression 7 minimizes the imbalance of supply and demand across regions. If the load is perfectly balanced, Expression 7 is zero. Expression 8 minimizes region oveload. Without overload, Expression 8 is a constant, $\text{sigmoid}(0) = 0.5$. As the region becomes more severely overloaded (i.e., demand above supply), Expression 8 approaches 1, and the attempt to minimize Expression 8 would lead to reducing overload.

Expressions 3 and 9 *soft-balance* the supply and demand of storage space across regions. However, there is no term similar to Expression 8 to minimize storage space overload, as it is disallowed to allocate more space than available, enforced by Expression 4. Similarly, Equation 5 ensures that every region's CPU demand is below supply. Note that there is no such hard constraint for GPUs, as it would often lead to an unsolvable problem due to GPU scarcity.

Expression 6 only applies to time-sensitive high-priority jobs, ensuring that most of these jobs are trained in their home regions, i.e., satisfying the *training-at-home-region* property. Moreover, we empirically validate but do not mandate that the number of jobs satisfying this property does not decrease over time.

## 3.4 Efficient Approximate Solution

The MIP problem in Figure 4 is NP-hard. Tetris uses a hill-climbing algorithm [26] to compute an efficient approximate solution. Starting from the current placement, Tetris goes over each table and finds the best region for it out of different home region choices. The best region should have the lowest cost (Expressions 1-3) and can satisfy all constraints (Expressions 4-6). If the best region differs from the current home region of the table, the table is added to a move queue. The queue is ordered by the gain of table moves, i.e., the initial cost before the move minus the new cost after the move.

After determining the new home region for all tables, Tetris iterates over the queue to move tables, starting from the table with the highest move gain. Before moving a table, it recalculates its best region because moving tables in prior iterations may have changed the cost of moving this table. The moves continue until either the queue becomes empty or a daily move quota is reached due to limited cross-region network bandwidth. Tetris runs this algorithm daily which takes about five hours to complete, although the actual data replication may take much longer to finish.

To move a table, Tetris replicates the table to the new home region before deleting its data in the old home region. A data-migration service schedules and executes cross-region data replication. Tetris can set a soft deadline for a migration operation, and the migration service allocates the necessary bandwidth accordingly. Our network's cross-region traffic receives differentiated quality of service to ensure that background bulk data replication does not affect latency-sensitive services. As for the data-replication size, the P50 (50th percentile) is 565MB, the P90 is 103GB, and the P99 is 7.5TB. In terms of data transfer time, the P50 is 2.1 hours, the P90 is 3.7 hours, and the P99 percentile is 4.9 hours.

It is possible that the new home region becomes suboptimal during the move due to various reasons. Typically, the next day's rerun of Tetris will correct the problem. However, if MAST needs to schedule the corresponding job before the rerun, it can use heuristics to determine whether on-demand data movement is needed to temporarily fix the issue.

In addition to the hill-climbing algorithm, we have evaluated other solutions such as commercial MIP solvers [12, 18], but a comprehensive comparison with local-search alternatives is yet to be conducted. We chose hill-climbing for several reasons. Firstly, it offers greater scalability than other solutions we have explored. Secondly, hill-climbing simplifies the definition of the MIP problem formulation. As mentioned in §3.3, $R(\text{table}_i)$ is the sole decision variable, with $R(\text{job}_j)$ inferred from $R(\text{table}_i)$, and GPU and storage demands per region further inferred from $R(\text{job}_j)$. In traditional MIP formulas, $R(\text{job}_j)$ must also be declared as a decision variable constrained by $R(\text{table}_i)$, and the same must be done for all demand requirements. This would lead to mathematical formulas much more complex than those shown in §3.3. In contrast, with hill-climbing, we can compute $R(\text{job}_j)$ and demand requirements from $R(\text{table}_i)$ using code, which offers more flexibility. Lastly, the outputs of hill-climbing are interpretable and easy to debug, as we understand the exact reasons to migrate tables in each iteration. Given that hyperscale ML training is still relatively new and the problem formulation continues to evolve, the interpretability and debuggability of hill-climbing present significant advantages.

## 3.5 Creating Extra Table Replicas

In addition to a table's home region, it may have replicas in other regions for various reasons. For disaster recovery (DR) purposes, each table has a replica outside of its home region. The region that stores the DR replica is selected based on a DR policy, which takes into account factors such as the probability of correlated regional failures and the availability of hardware to handle increased load after a disaster.

After the hill-climbing algorithm finishes, some ML jobs may not be able to run in any region, because their home region $R(\text{job}_j)$ is NULL or does not have the type of GPUs needed by the job. The DR copy may fix some of these jobs, and for the remaining ones, Tetris creates additional replicas of the tables accessed by them in other regions, allowing them to be scheduled there. For example, suppose $\text{table}_i$ is accessed by $\text{job}_1$ and $\text{job}_2$, which require $\text{GPU}_1$ and $\text{GPU}_2$, respectively. However, $\text{GPU}_1$ and $\text{GPU}_2$ are not available together in any region. Thus, Tetris may set $R(\text{table}_i)$ to be a region that has $\text{GPU}_1$ to run $\text{job}_1$ and then create a replica of the table in a region that has $\text{GPU}_2$ to run $\text{job}_2$. If multiple regions have $\text{GPU}_2$, Tetris selects the region with the highest supply.

A small fraction of tables, known as hot tables, are accessed by a large number of jobs (Figure 3). If these tables are replicated to only a few regions, it may create a significant load imbalance, as a large number of jobs dependent on these tables will be forced to run in a few regions. To address this problem, Tetris widely replicates hot tables per GPU type.

|  |  | Scope of job queue management |  |  |
|---|---|---|---|---|
|  |  | Cluster | Regional | Global |
| **Scope of resource allocation** | Cluster | (1) Borg, Hydra, Yugong | (2) | (3) |
|  | Regional | (4) ✗ | (5) | (6) MAST |
|  | Global | (7) ✗ | (8) ✗ | (9) "Ideal" |

Table 1: Design space partitioned by the scheduling scope. The symbol ✗ indicates invalid solutions.

For each region and each GPU$_j$, Tetris computes the region's storage quota for GPU$_j$ based on the supply of GPU$_j$. Tetris sorts tables accessed by jobs using GPU$_j$ based on the tables' hotness, and replicates as many hot tables as possible until it reaches the storage quota for GPU$_j$.

Extra table replicas provide MAST's fast path with greater flexibility to choose the region for hosting a job at runtime. However, replicating a massive amount of data across regions could take hours, delaying the start of some time-sensitive jobs. To address this problem, Tetris takes a combination of measures. First, Expression 6 enforces that sufficient jobs are trained in their home regions. Second, instead of replicating the whole table, Tetris can be configured to only replicate the partitions needed by the training job. Finally, Tetris prioritizes first replicating data needed by high-priority jobs to meet their deadlines.

Overall, the additional table replicas created by Tetris increase storage consumption by approximately 75% to 125%. However, given the high cost of GPUs, we deem this a justifiable trade-off.

## 4 Fast-path Job Scheduling

While the slow path asynchronously prepares data for ML training, the fast path schedules ML workloads in real-time. Before presenting MAST's scheduling solution, we explore the design space to understand its rationale.

### 4.1 Exploring the Scheduling Design Space

Traditionally, the federation approach [10, 20, 28] employs *early binding*, dispatching a new job to a cluster based on the current estimated cluster load, even if the cluster has no available resources to run the job immediately. In contrast, MAST adopts *late binding*, dispatching a job to a cluster only when certain that the cluster has available resources to run the job immediately.

Moreover, traditional scheduling systems handle all scheduling functions at the cluster scope: job queue management, resource allocation, and container orchestration. Below, we explore solutions that manage job queues and resource allocation at the regional or global scope.

**Comparing solutions.** Table 1 shows the solution space. Solution (1) is the traditional approach, with Borg [48] as an example, where job queue, resource allocation, and container orchestration are all managed at the cluster scope. Solution (6) is our approach, where the job queue is managed at the global scope and resource allocation is managed at the regional scope. Solution (9) is the "ideal" approach, where both the job queue and resource allocation are managed at the global scope. With a global view of all jobs and machines, theoretically, it can achieve optimal job placement, but the limited scalability of this approach is a main shortcoming.

For ML training workloads, it can be proven that among algorithms that schedule one job at a time, MAST achieves the same optimal job placement as (9) due to several reasons. First, all tasks of a training workload must be allocated to the same region for locality, simplifying resource allocation calculations to within regions. Moreover, following the *exhaustive-search* principle, MAST computes a resource allocation plan for the workload in *every* region with training data, and then chooses the best plan for execution. As a result, it can achieve optimal placement for individual workloads. However, solution (9) has an advantage over MAST in jointly optimizing the placement of a set of workloads. For instance, after MAST places *workload 1* in region $X$, it may discover that no region can accommodate *workload 2*. In contrast, solution (9) may intentionally place *workload 1* in region $Y$ and leave region $X$ to handle *workload 2*.

Solutions (2), (3), and (5) improve upon solution (1) as they can either better balance the load or manage allocation at a larger scope, but they still cannot provide the same level of scheduling quality as (6). Solutions (4), (7), and (8) are invalid as their scope of resource allocation is bigger than their scope of job queue management. Among the solutions in Table 1, assuming (9) is not scalable, our preference is (6) > (3) > (5) > (2) > (1).

**Federated systems.** Hydra [10] and Yugong [20] use the federated approach. Both fall under solution (1) as they employ early binding of a job to a cluster, and then manage the job queue and resource allocation with the cluster. This approach aligns well with the nature of lightweight analytics jobs, the focus of Hydra and Yugong. Such jobs demand high scheduling throughput but typically do not involve complex decisions like job preemption.

### 4.2 Global ML Scheduler (GMS)

Adhering to the *scope-decoupling principle* and as illustrated in Figure 1, MAST splits the scheduling responsibilities among different components: the Global ML Scheduler (GMS) manages the global job queue, the Regional ML Scheduler (RMS) allocates regional resources, and the Cluster Manager (CM) is responsible for container orchestration.

The main responsibility of GMS is to select the next workload to schedule among all pending workloads in the global job queue. For each pending workload, GMS calculates a $<priority, credit>$ tuple. It schedules the workload with the highest *priority* and, in case of a tie, selects the one with the highest *credit* for scheduling.

The *priority* is affected by quota usage. Each workload belongs to a tenant, i.e., a team. Tenants are assigned a priority level and a quota for running their workloads. The quota specifies the maximum number of GPUs and CPUs that a tenant can use simultaneously. Workloads from a tenant that has not exhausted its quota are assigned the tenant's priority, categorized as *within-quota workloads*. Conversely, when a tenant has used up its quota, its workloads are assigned the lowest priority and categorized as *over-quota workloads*. These workloads run opportunistically and can be preempted when a higher priority workload arrives. Tenant priorities are manually assigned based on business priorities. The strict adherence to these priorities does result in preemption, which is the intended effect. Currently, MAST uses seven priority levels. The distribution of workloads across these priorities (from highest to lowest) is as follows: 3%, 20%, 16%, 54%, 0.2%, 0.5%, and 0.02%. The remaining 6% of workloads do not specify a priority and are consequently treated as the lowest priority.

The *credit* of a workload is calculated as follows. Intuitively, a workload $W$ has a higher *credit* if it has been waiting longer (Expression 10) or belongs to a tenant that has used fewer resources than others (Expression 11).

$$\text{credit(L)} = w_{\text{workload\_age}} \times \min(\text{L.age}, C_{\text{age\_cap}}) \tag{10}$$

$$+ w_{\text{fair\_share}} \times (1 - \frac{\text{window\_avg(L.tenant.resources\_used)}}{\text{window\_avg(all\_tenants.resource\_used)}} \tag{11}$$

Here, $L$ represents a workload, $C_{\text{age\_cap}}$ is a constant, and $w_{\text{workload\_age}}$ and $w_{\text{fair\_share}}$ are tunable weights.

To determine the $<priority, credit>$ tuple for a workload, GMS calculates each workload's *credit* and sorts pending workloads based on it. It scans them to assess if they can be scheduled within their tenant's quota. GMS maintains a *resource_used* variable for each tenant, initialized to include resources used by the tenant's running workloads. When scanning a pending workload $W$, GMS checks if adding $W$'s resource requirement to *resource_used* would exceed the tenant's quota. If so, $W$ is assigned the lowest priority level; otherwise, $W$ inherits its tenant's priority level, and GMS updates the tenant's *resource_used* to include $W$'s resources.

Periodically, GMS executes a *GMS-scan pass* to update $<priority, credit>$ tuples for all pending workloads. This approach is chosen because the state change of one workload may impact others' $<priority, credit>$. Specifically, a workload's *credit* is influenced by other tenants' resource usage (Equation 11). Additionally, a workload's *priority* is tied to its tenant's other workloads' resource usage. This updating-all-workloads strategy enables MAST to implement sophisticated quota and priority management policies. The scalability of GMS with this approach depends on the frequency and duration of the GMS-scan pass.

### 4.2.1 Scalability of GMS

In practice, GMS scales well for ML workloads due to several factors. First, ML training workloads, running for extended periods on many machines, necessitate higher-quality scheduling decisions but lower scheduling throughput compared to short-lived batch jobs [10, 38, 41, 50]. Second, GMS has minimal responsibilities, calculating <priority,credit> for each pending workload and storing it in the job-queue database. Placement plans are computed by RMS, not GMS. Third, our evaluation in §5.4 indicates that the current GMS implementation can support workload growth by a factor of 8.8. Finally, currently implemented in Python for simplicity, if it becomes a bottleneck, we plan to scale it further by a factor of 10-100 by switching to C++ and parallelizing computation for different tenants.

## 4.3 Regional ML Scheduler (RMS)

RMSs perform auctions in a distributed manner to schedule ML workloads. Each RMS constantly checks the job-queue database maintained by GMS, and attempts to schedule the ML workload with the highest $<priority, credit>$.

To schedule a workload, an RMS consults a real-time component of Tetris to check whether its local region has the required data and necessary hardware types. If not, the RMS abandons the auction. If all RMSs abandon the auction, potentially occurring with the first-time execution of a new workload, MAST will start data replication, waiting for its completion to ensure some regions have both the necessary data and hardware.

Typically, due to Tetris, multiple regions have the required data and hardware types for the workload. Following the *exhaustive-search* principle, in each such region, RMS calculates a placement plan for the workload along with a corresponding placement-quality score ($P_{score}$). RMSs engage in an *auction* to identify the RMS with the highest $P_{score}$, which will execute the workload. If no region can generate an immediate placement plan for the workload, it enters the *waiting* state.

A region may have multiple ML dynamic clusters, and the workload can comprise multiple jobs, each assignable to a different cluster. Adhering to the *exhaustive-search* principle, for each job in the workload, RMS calculates a placement plan and $P_{score}$ for every ML dynamic cluster in the region. It chooses the cluster with the highest $P_{score}$ to host the job. The overall $P_{score}$ for the workload is determined by summing up the $P_{score}$ for each job in the workload.

When comparing two placement plans for a job, the one with a higher $P_{score}$ wins. A plan has a higher score if it uses available resources to run the new job without preempting any running jobs. If preemption is necessary, a higher score is achieved by preempting jobs with lower priority, fewer jobs, or those running for a longer duration. The last condition implies refraining from preempting newly started jobs.

To generate a placement plan for a job in an ML dynamic cluster, RMS checks if it can allocate the job using available resources without preempting running jobs. For enhanced task locality within the same job, RMS sorts available machines based on rack IDs, allocating machines in the same or nearby racks in batches. While scanning, RMS aims to use the minimum number of machines by prioritizing those with the highest available CPU/GPU resources.

If preemption is necessary, RMS prioritizes preempting lower-priority jobs. It sorts all running jobs based on priority and running time, initiating the scan with the job having the lowest priority and longest running time. The scanning process continues until sufficient resources are found to execute the new job, combining available resources with those to be released through preemption.

RMS minimizes preempted jobs. For example, if a new job needs 10 GPUs and the scan reveals options of 4, 5, and 12 GPUs by preempting $\text{job}_1$, $\text{job}_2$, and $\text{job}_3$ respectively, the optimal choice is preempting $\text{job}_3$ without affecting $\text{job}_1$ or $\text{job}_2$. To achieve this, RMS re-sorts preempted jobs by size, prioritizing larger jobs first.

Multiple optimizations improve RMS performance, with the most effective being the use of a *negative cache*. When RMS cannot allocate resources for a job, it saves the decision in the cache. If it later attempts to allocate a job of the same or larger size, the cache signals that the allocation will fail. Specifically, the negative cache is initialized at the start of every GMS-scan pass that examines all pending jobs, and is cleared after the GMS-scan pass finishes. It is implemented as a hashtable that stores the scheduling properties of the jobs that could not be scheduled during the GMS-scan pass. These scheduling properties encompass the number of GPUs and CPU cores requested by the job, memory, hardware type, and so on. Such information consumes little memory, and there is no need for cache eviction during a GMS-scan pass. Overall, the negative cache is highly effective as unsuccessful placement attempts far outnumber successful ones due to nearly constant full resource allocation. It filters out the majority of these unsuccessful attempts early on, thanks to a cache hit rate of about 80% in practice.

### 4.3.1 Scalability of RMS

RMS demonstrates sufficient scalability, as evidenced by the analysis below. Scalability is examined concerning the number of regions ($r$) and the amount of ML hardware per region ($h$). Adding more regions does not increase the computation load of RMS, as it schedules workloads only for data stored in the respective region. When $h$ remains constant, the number of such workloads also remains unchanged. However, linear growth in $h$ results in quadratic growth in RMS's computation load. The complexity of RMS's exhaustive search is $E(h) = O(D(h) \times J(h))$, where $D$ is the number of ML dynamic clusters and $J$ is the number of jobs scheduled on these clusters. As both $D$ and $J$ are proportional to $h$, $E(h)$ experiences quadratic growth.

Our evaluation in §5.4 demonstrates that RMS can handle a 12x increase in $h$ compared to our current production load. This scalability is likely sufficient even for the long run, given that the growth of $h$ is constrained by the fixed electricity supply of a datacenter region. Currently, the largest RMS manages around 20 dynamic clusters, comprising a total of 64,000 CPU machines and 20,000 GPUs. In the improbable scenario of RMS becoming a bottleneck, we plan to parallelize scheduling for non-conflicting workloads with training data in non-overlapping regions. Additionally, if needed, RMS can be sharded to scale out, with each shard handling a subset of ML dynamic clusters.

## 4.4 Cluster Manager

Our cluster manager (CM), Twine [44], has the distinguishing feature of managing a dynamic cluster whose machine membership may be continuously updated by RAS [36]. The CM instances managing ML and non-ML clusters are separate and do not interfere with each other, while RAS can dynamically move machines between them to avoid stranded capacity.

We choose not to use a single, generic cluster to handle mixed ML and non-ML workloads, as it is suboptimal for our large-scale operations. Our large fleet size necessitates partitioning machines into independent clusters for effective management. Combining ML and non-ML workloads in a cluster compromises optimization for either type, whereas our scale benefits significantly from workload-specific optimization. For instance, online services prefer spreading across fault domains, whereas ML training workloads prefer not to be spread widely for better network performance. Furthermore, as gang jobs, ML training workloads prefer, for example, 10 out of 100 jobs to fail entirely while the remaining 90 jobs continue, as opposed to each job experiencing a 10% task termination. Optimizing for spread would lead to the latter undesirable situation. Previously, CM handled these complex differences between ML and non-ML workloads on the fast path of real-time job scheduling, often resulting in suboptimal choices due to limited computation time. Consequently, we adopted the strategy of RAS running on the slow path to pre-built separate ML and non-ML dynamic clusters that are deeply optimized for respective workloads, while simplifying the responsibilities of CM on the fast path.

CM and RMS collaborate in managing workloads. For example, when new machines are added to an ML dynamic cluster, the corresponding CM notifies RMS of this change. With complete information cached in its memory, RMS can efficiently compute placement plans. When a region's placement plan is selected as the best plan for execution, its RMS directs the corresponding clusters' CMs to execute the plan and run the relevant jobs. This may require preempting running jobs and checkpointing their current status, initializing containers for the new jobs, and restoring their job states if they were previously preempted.

## 4.5 Fault Tolerance

GMS, RMS, and CM are fault-tolerant and highly available, operating in a leader-follower mode. Specifically, two instances of GMS run in different regions, two instances of RMS operate in the same region, and three instances of CM serve the same cluster. They all follow a stateless design, storing their persistent state in a shared and replicated database. In the event of a leader failure, the follower can reconstruct its state from the database and the lower-level component (i.e., GMS from RMS and RMS from CM).

## 4.6 Limitations

In our hyperscale production environment, we prioritize implementation simplicity and robustness, leading to some implementation limitations rather than inherent design flaws. One such limitation is the inability to distribute a job's tasks across different clusters, despite the capability to allocate workload jobs to various dynamic clusters. RMS can compute a placement plan for distributing a job's tasks across different clusters by leveraging its comprehensive view of all resources in the ML clusters within the region. However, the integration with our cluster manager [44]'s "virtual job" feature, crucial for effectively managing scattered tasks as one virtual job, is not yet implemented.

Another implementation limitation is that currently, RMS schedules only one ML workload at a time. While it parallelizes the computation of placing multiple jobs from one workload into different clusters, it does not commence scheduling the next workload until a decision has been made for the current one. This simple approach is used because it is adequate and still has headroom to support further growth, as discussed in §5.4. However, if a bottleneck arises in the future, we are prepared to transition to scheduling multiple workloads in parallel. Moreover, scheduling multiple workloads simultaneously presents opportunities for enhancing scheduling quality. Note that although the current implementation schedules one ML workload at a time, different workloads can still run in parallel. Once a workload X is dispatched to run, without waiting for X to finish execution, the scheduler immediately schedules the next workload Y.

# 5 Evaluation

Our evaluation attempts to answer the following questions:

1. What are the important ML workload statistics?
2. Can MAST achieve a high resource allocation rate?
3. Is Tetris effective in ensuring colocation of data and compute resources?
4. Are GMS and RMS sufficiently scalable?
5. How long does it take to schedule a workload?
6. How does MAST compare with alternative solutions?

## 5.1 ML Training Workload Statistics

Currently, MAST is scheduling tens of thousands of ML training workloads daily across tens of regions, consuming $O(100,000)$ GPUs and $O(100,000)$ CPU machines. About 70% of the workloads use GPUs for training, while the remaining use CPUs for training. About 30% of the workloads are recurring, while the remaining are first-time workloads. On average, each workload gets preempted once, which leads to about 100,000 scheduling attempts daily. High-priority workloads may never get preempted and low-priority workloads may get preempted multiple times.

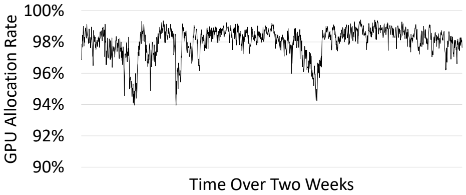


Figure 5: GPU allocation rate.

In terms of the number of machines utilized by a workload, the values for the 50th, 90th, and 99th percentiles are 72, 180, and 205, respectively. As for the number of GPUs used by GPU-consuming workloads, the 50th, 90th, and 99th percentiles are 16, 64, and 128, respectively, and the largest workload today, LLM pre-training, uses tens of thousands of GPUs. We measure the duration of workloads in terms of their execution time until the subsequent preemption, as it is more pertinent for a scheduling system. For GPU workloads, the 50th, 90th, and 99th percentiles of the duration are 20 minutes, 6.7 hours, and 66 hours, respectively. As for CPU workloads, the corresponding percentiles are 38 minutes, 7.9 hours, and 38 hours. Overall, training workloads run on many machines for an extended period. Therefore, it is worthwhile to spend time computing high-quality scheduling decisions.

## 5.2 Effectiveness of Global ML Scheduling

Thanks to the flexibility of placing both data and workloads globally, MAST has achieved a high average *allocation rate* of 98% for its GPU machines, as shown in Figure 5. The 2% loss is due to factors such as the overhead of preemption, inherent latency of scheduling, and imbalanced data and GPU distribution across regions. The allocation rate is determined by dividing the total hardware hours allocated to workloads by the total available hardware hours. Note that the GPU allocation rate differs from GPU utilization because even if some GPUs are allocated to a workload, they may be underutilized due to various factors, such as the workload's internal communication bottlenecks. We use the allocation rate as the metric because it is more relevant for a scheduling system, which is the primary focus of this paper, while GPU utilization is more pertinent for the ML training framework.

In comparison, the allocation rate for CPU machines is lower. Dedicated CPU machines for ML workloads have an allocation rate of 87%, while elastic CPU machines, which are borrowed temporarily from non-ML workloads, have an allocation rate of 72%. The lower allocation rate for CPU machines is largely due to slight overprovisioning to guarantee that costly GPU machines are never left idle due to a lack of available CPU machines to work with.

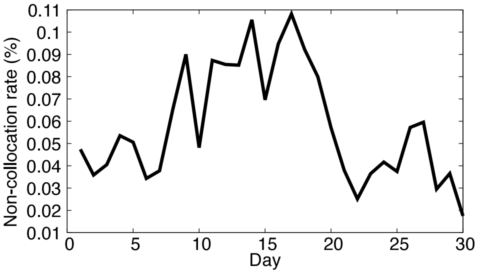


Figure 6: Percentage of workloads that have to wait for data.

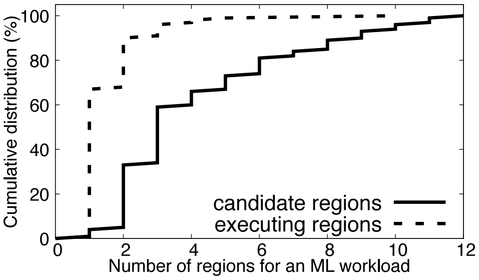


Figure 7: Cumulative distribution of candidate and executing regions per workload over 30 days.

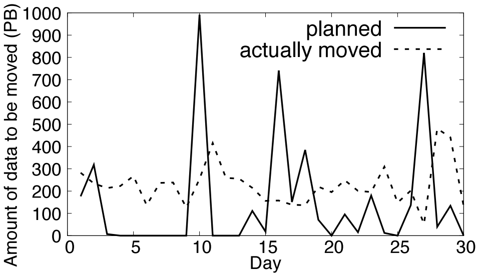


Figure 8: Daily data movement by Tetris.

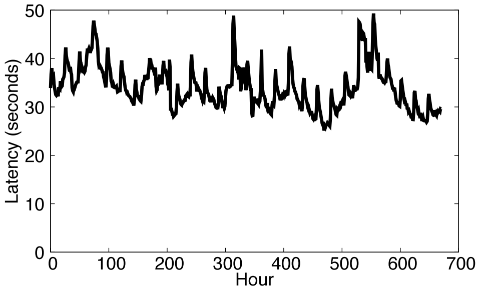


Figure 9: The latency of a GMS-scan pass in GMS (each data point is the average latency within one hour).

## 5.3 Effectiveness of Data Placement

Figure 6 shows the percentage of workloads that cannot satisfy the data-GPU-collocation property (§3.1) and thus have to wait for on-demand data movement to complete. The non-collocation rate is usually below 0.1%, demonstrating the effectiveness of Tetris. Although Tetris creates extra table replicas to increase the collocation rate (§3.5), non-collocation may still occur due to a workload appearing for the first time or due to machine maintenance rendering certain data or hardware unavailable. Considering 70% of our workloads are first-time ones, this figure shows most of them can still benefit from the data placement planned for recurring workloads.

Thanks to Tetris, often multiple regions can host the same workload, which gives MAST the flexibility to migrate the workload across regions on different days. In Figure 7, the "*candidate regions*" have the required training data and hardware types to host a workload, while the "*executing regions*" have actually hosted the workload on different days. As shown in the figure, the majority of workloads have two or more candidate regions, and approximately 40% of them have four or more. The number of executing regions per workload is smaller; roughly 30% of the workloads have more than one executing region, indicating they are relocated across regions during this period. This demonstrates that global ML scheduling indeed works as intended to dynamically optimize workload placement across regions.

Figure 8 shows the amount of data that Tetris move daily. The spikes in the planned movement are due to onboarding a new workload, which caused a large amount of data to be moved across regions. However, the data migration service has a limit on the maximum amount of data moved per day. Therefore, the actual amount of data moved per day is flatter. This figure shows that Tetris proactively moves hundreds of petabytes of data across regions daily. This enables the fast path to more easily colocate computation with data, and achieve the high GPU allocation rate of 98%.

Tetris's hill climbing algorithm runs daily on a single machine and it typically takes about five hours to finish. Although its CPU utilization is not very high, it spends a significant amount of time on I/Os to fetch various metadata necessary for computing the data placement plan. Currently, the performance of Tetris is not a major bottleneck, and can be further optimized as needed.

## 5.4 Scalability of GMS and RMS

**Scalability of GMS.** GMS periodically computes the $<priority, credit>$ tuples for all workloads (§4.2). We refer to one round of such computation as a *GMS-scan pass*. The scalability of GMS depends on the frequency and latency of the GMS-scan pass. Although its theoretical complexity is $O(N \log N)$ due to sorting, where $N$ is the number of workloads, its actual execution time is approximately $O(N)$, dominated by the sequential computation of $<priority, credit>$ for each workload. Figure 9 shows the average latency of the GMS-scan pass. On average, it takes approximately 34 seconds for GMS to rank 6,000-10,000 workloads. Our evaluation shows that running the GMS-scan pass once every 5 minutes still produces high-quality scheduling. This implies that GMS can support $\frac{5 \text{ minutes}}{34 \text{ seconds}} = 8.8$ times more workloads. As described in §4.2.1, if needed, we can scale the GMS by another factor of 10-100 by switching from Python to C++ and parallelizing its computation for different tenants.

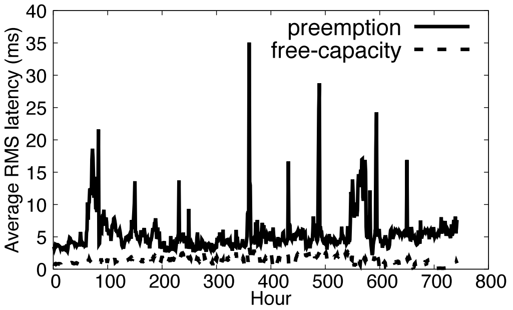


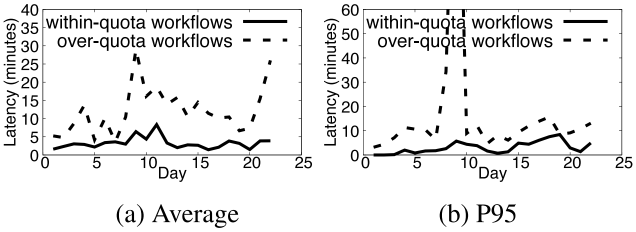
Figure 10: Average latency of scheduling a workload at the slowest RMS. It is computed by taking the maximum of the average latencies computed at each RMS every minute.

**Scalability of RMS.** Since our current RMS implementation does not initiate scheduling of the next workload until a decision has been made for the current one (§4.6), the maximum throughput of RMS can be estimated from its latency to schedule a single workload. As shown in Figure 10, the slowest RMS takes approximately 1.3ms and 5.5ms to schedule a workload without and with preemption, respectively. Even assuming that all scheduling requires preemption, the RMS can schedule 24*3600*1000/5.5=15 million workloads per day. Considering the current throughput of about 100K scheduling attempts per day and the quadratic growth of the computation load caused by exhaustive search (§4.3.1), the RMS can support approximately $\sqrt{\frac{15M}{100K}}$=12x more workloads and 12x more ML hardware. See §4.3.1 for a discussion on how to further scale RMS.

## 5.5 Scheduling Latency

Figures 11(a) and (b) show the average and P95 latency, respectively, for pending workloads to enter the running state, including queuing delay, scheduling-algorithm run time, and preemption time but not workload execution time. The preemption time is a major factor in the total delay. If workload A preempts B, rescheduling B counts as a new scheduling event in these figures, starting from the time B is preempted to the time B runs again. The preemption time of B counts towards A's latency as A needs to wait for the preemption to finish. One primary service level objective for MAST is P95 latency. These figures show that MAST has maintained consistently low latency for within-quota workloads. However, for over-quota workloads, the latency can occasionally be erratic, depending on the workload mix. This emphasizes the importance of distinguishing between within-quota and over-quota workloads.

To start a workload, MAST needs to acquire containers from the cluster manager and set up all containers. The P50 latency of this step is 150 seconds, while the P90 is 278 seconds, and the P99 is 449 seconds. For massive LLM jobs, the whole process could take 10 minutes or longer.



(a) Average

(b) P95

Figure 11: Latency for a pending workload to start running.

## 5.6 Comparison with Alternative Solutions

**Tetris.** The closest work to Tetris is Alibaba's Yugong [20], which uses MIP to place data for analytics jobs based on CPU (but not GPU), storage, and network constraints. It enforces hard quotas and would not produce a solution when resources are insufficient. However, as GPUs are scarce in our environment with consistently higher demand than supply (§3.2), Yugong would never provide a solution.

Since Yugong is not comparable to Tetris, we evaluate Tetris's different versions to demonstrate the importance of its key features. In the initial stage (V0) of MAST, users manually selected regions for table placement, and tables could not migrate across regions. In 2022, V1 was developed, which automated data placement, aligning with the approach in Section 3, albeit with key distinctions. V1 was a major improvement over V0, reducing the GPU demand-over-supply ratio for high-priority jobs in the most overloaded region from 2.63 to 0.98, with the standard deviation dropping from 0.76 to 0.30. Moreover, it boosts the training-at-home-region rate from 90.82% to 93.02%.

Despite V1's success, various limitations prompted the development of V2. As described in § 3.3, key differences include V2 penalizing GPU overload more than penalizing underload, and incorporating *soft-balance* alongside *hard-quota* for storage. Additionally, V1 defined the collocation rate uniformly for all workloads, treating small and large, high-priority and low-priority workloads alike. Recognizing the practical importance of large high-priority workloads, V2 has revised this definition to be the demand (i.e., GPU hours) of high-priority workloads that can satisfy colocation. With these enhancements, V2 significantly increased the collocation rate under the new definition from 78% to 96% compared to V1.

**Fast-path Scheduler.** We built a simulator to evaluate different scheduling algorithms of the fast path. The simulator takes a trace of past workloads as input, and follows the same logic of MAST's fast-path scheduler, excepts that it does not actually execute a workload, but instead assumes its running time is the same as recorded in the trace. For comparison, we modified the simulator to implement a federated approach, which dispatches a workload to the region with the lowest demand over supply rate of the required GPU type. When playing an 8-hour trace to the simulator, we find that for MAST, the rate of workloads violating SLOs stays below 1.3% all the time. For the federated approach, however, the SLO violating rate is much higher, especially when load is high. High-priority workloads suffer more, reaching a 50% SLO violating rate during busy hours. This is because high-priority workloads often need to preempt low-priority ones to guarantee SLOs, but with a simple heuristic, the federated approach may not be able to dispatch a high-priority workload to the right region where it can preempt others.

## 6 Discussions

**Solutions suitable for smaller organizations.** While MAST is designed for hyperscalers, some of its principles can be applied to smaller organizations. If a small organization only needs to run training workloads in a single cluster, implying one region, then it does not need any of MAST's advanced capabilities. However, in the event of a power or network outage affecting the region, it would be unable to run any ML training jobs due to the lack of disaster-recovery capability.

If an organization's infrastructure operates in at least two regions to be disaster ready, then it can leverage the key ideas in MAST. Without MAST, they would default to solution (1) in Table 1, leading to suboptimal resource allocation due to the isolated operation of the two regions. With insights from MAST, if the scale of their infrastructure is small enough to be handled by a single resource allocator, they could adopt solution (9), yielding optimal placement results. If they do not want to significantly modify their cluster manager like Kubernetes [27], they could at least adopt solution (3), which involves a relatively minor change to use a global job queue but still offers significant benefits in balancing the load across regions. Furthermore, if they cannot afford to replicate every table across every region, they would need a component similar to Tetris to intelligently determine data placement.

**Future work.** Currently, Tetris considers storage quota and network bandwidth as hard constraints. It is valuable to understand the impact of adding storage and cross-region network bandwidth to better utilize expensive GPUs.

For scalability, Tetris currently determines home regions (§3.2 to §3.4) and creates additional table replicas (§3.5) in separate steps. We plan to explore whether there exists an efficient MIP problem formulation that can simultaneously determine the optimal number of replicas and the home region for each table. It is important that such a problem formulation can be solved in a scalable manner.

Finally, we plan to leverage the fact that the slow path (Tetris) not only determines data placement but also, as a byproduct, calculates the placement of recurring jobs. The latter is currently overlooked by the fast path when placing jobs. Because unexpected one-time jobs may make some of the slow path's job placement decisions unfeasible or suboptimal, future research is needed to better connect the fast and slow paths. This would allow us to benefit from the job-placement decisions that the slow path has more time to compute.

## 7 Related Work

**Scheduling within a cluster.** There are many prior works about scheduling within a single cluster, including general-purpose schedulers [14,16,19,22,38,47,56] and those specific to ML training [1,2,5–7,9,13,17,21,23–25,30,31,33–35,39,40,45,49,51–54,57,59]. The latter often considers specific characteristics of ML training, such as model accuracy, gang scheduling, sensitivity to network topology, heterogeneity of compute resources, elasticity, etc. Cluster-level scheduling is largely orthogonal to the design principles of MAST, though some of their ideas may be applicable in the cluster-level placement algorithm of MAST.

**Scheduling across clusters.** Among existing systems, Yugong [20] and Hydra [10] are the closest to MAST as they can schedule jobs across cluster or datacenters, but they still differ from MAST since 1) they perform early-binding at cluster scope (§4.1), 2) they don't take GPU into consideration (§1 and §3), and 3) they rely on simple heuristics to dispatch jobs to clusters. Finally, in the data warehouse hierarchy (hundreds of namespaces→millions of tables→ billions of data partitions), Yugong places data at the namespace level (called "projects" in the paper), whereas Tetris places data at the partition level, which provides more opportunities for fine-grained optimization. However, Tetris' fine-grained placement makes the optimization problem about $10^6$ times larger.

Singularity [43] is the only ML-specific global-scale scheduler we are aware of. However, the article does not disclose details about the scheduling part, but focuses more on how to provide elasticity to ML training jobs, and thus it's impossible for us to provide a concrete comparison.

## 8 Conclusion

This paper demonstrates that by utilizing the three design principles of *temporal decoupling*, *scope decoupling*, and *exhaustive search*, we can build a global ML training scheduler that can 1) scale to tens of regions and hundreds of thousands of machines and 2) provide high-quality data and job placement to achieve almost 100% allocation of GPUs.

## Acknowledgments

This paper presents years of work by past and current members of several teams at Meta, including MAST, Tetris, and Twine [44]. In particular, we would like to call out some current team members not on the author list: Mike Begic, Rick Chang, Cheng Cheng, Fuat Geleri, Ankit Gureja, Christian Guirguis, Hamid Jahanjou, Johan Jatko, Jonathan Kaldor, Norbert Koscielniak, Alexander Kramarov, Mengda Liu, Runming Lu, Duy Nguyen, Ivan Obraztsov, Colin Ovens, Marcin Pawlowski, Nader Riad, Derek Shao, Ahmed Sharif, Mihails Smolins, Bhushan Sonawane, Peeyush Taneja, Yingjie Tang, Hai Dang Tran, Tom Wan, Anthony Wang, Runmin Wang, Wenbang Wang, and Lukasz Wesolowski. We thank all reviewers, and especially our shepherd, Ryan Huang, for their insightful comments.

## References

[1] Saurabh Agarwal, Chengpo Yan, Ziyi Zhang, and Shivaram Venkataraman. Bagpipe: Accelerating deep recommendation model training. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 348–363, New York, NY, USA, 2023. Association for Computing Machinery.

[2] Marcelo Amaral, Jordà Polo, David Carrera, Seetharami Seelam, and Malgorzata Steinder. Topology-Aware GPU Scheduling for Learning Workloads in Cloud Environments. In *SC17: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2017.

[3] AWS Regions, 2023. https://aws.amazon.com/about-aws/global-infrastructure/.

[4] Azure Regions, 2023. https://azure.microsoft.com/en-us/explore/global-infrastructure/products-by-region/.

[5] Yixin Bao, Yanghua Peng, and Chuan Wu. Deep Learning-based Job Placement in Distributed Machine Learning Clusters. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 505–513, 2019.

[6] Yixin Bao, Yanghua Peng, Chuan Wu, and Zongpeng Li. Online Job Scheduling in Distributed Machine Learning Clusters. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 495–503, 2018.

[7] Zhengda Bian, Shenggui Li, Wei Wang, and Yang You. Online Evolutionary Batch Size Orchestration for Scheduling Deep Learning Workloads in GPU Clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[8] Jesús Bobadilla, Fernando Ortega, Antonio Hernando, and Abraham Gutiérrez. Recommender systems survey. *Knowledge-based systems*, 46:109–132, 2013.

[9] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing Efficiency and Fairness in Heterogeneous GPU Clusters for Deep Learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[10] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M. Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, Roni Burd, Sarvesh Sakalanaga, Chris Douglas, Bill Ramsey, and Raghu Ramakrishnan. Hydra: a federated resource manager for data-center scale analytics. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 177–192, Boston, MA, February 2019. USENIX Association.

[11] Marius Eriksen, Kaushik Veeraraghavan, Yusuf Abdulghani, Andrew Birchall, Po-Yen Chou, Richard Cornew, Adela Kabiljo, Ranjith Kumar S, Maroo Lieuw, Justin Meza, Scott Michelson, Thomas Rohloff, Hayley Russell, Jeff Qin, and Chunqiang Tang. Global Capacity Management with Flux. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023.

[12] FICO Xpress Optimization. https://www.fico.com/en/products/fico-xpress-optimization.

[13] Wei Gao, Zhisheng Ye, Peng Sun, Yonggang Wen, and Tianwei Zhang. Chronus: A Novel Deadline-Aware Scheduler for Deep Learning Training Jobs. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 609–623, New York, NY, USA, 2021. Association for Computing Machinery.

[14] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N.M. Watson, and Steven Hand. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[15] Google Cloud Regions, 2023. https://cloud.google.com/blog/products/infrastructure/introducing-new-google-cloud-regions.

[16] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-Resource Packing for Cluster Schedulers. *SIGCOMM Comput. Commun. Rev.*, 44(4):455–466, aug 2014.

[17] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, Boston, MA, February 2019. USENIX Association.

[18] Gurobi Optimization. https://www.gurobi.com/.

[19] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA, March 2011. USENIX Association.

[20] Yuzhen Huang, Yingjie Shi, Zheng Zhong, Yihui Feng, James Cheng, Jiwei Li, Haochuan Fan, Chao Li, Tao Guan, and Jingren Zhou. Yugong: Geo-Distributed Data and Job Placement at Scale. *Proc. VLDB Endow.*, 12(12):2155–2169, aug 2019.

[21] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. Elastic Resource Sharing for Distributed Deep Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 721–739. USENIX Association, April 2021.

[22] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[23] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient Distributed Training of Large Models Using Pipeline Templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 382–395, New York, NY, USA, 2023. Association for Computing Machinery.

[24] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, Renton, WA, July 2019. USENIX Association.

[25] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 463–479, 2020.

[26] Donald L Kreher and Douglas R Stinson. Combinatorial algorithms: generation, enumeration, and search. *ACM SIGACT News*, 30(1):33–35, 1999.

[27] Kubernetes, 2020. https://kubernetes.io/.

[28] Kubernetes Federation, 2020. https://github.com/kubernetes/community/tree/master/sig-multicluster.

[29] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, Kaushik Veeraraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, and Chunqiang Tang. Shard Manager: A Generic Shard Management Framework for Geo-Distributed Applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 553–569, 2021.

[30] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proc. VLDB Endow.*, 13(12):3005–3018, aug 2020.

[31] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and Efficient GPU Cluster Scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, Santa Clara, CA, February 2020. USENIX Association.

[32] Yuan Mei, Luwei Cheng, Vanish Talwar, Michael Y Levin, Gabriela Jacques-Silva, Nikhil Simha, Anirban Banerjee, Brian Smith, Tim Williamson, Serhat Yilmaz, et al. Turbine: Facebook's Service Management Platform for Stream Processing. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1591–1602. IEEE, 2020.

[33] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Looking Beyond GPUs for DNN Scheduling on Multi-Tenant Clusters. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 579–596, Carlsbad, CA, July 2022. USENIX Association.

[34] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. Solving Large-Scale Granular Resource Allocation Problems Efficiently with POP. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 521–537, New York, NY, USA, 2021. Association for Computing Machinery.

[35] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498. USENIX Association, November 2020.

[36] Andrew Newell, Dimitrios Skarlatos, Jingyuan Fan, Pavan Kumar, Maxim Khutornenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, Brendon Daugherty, Apurva Samudra, Prashasti Baid, James

Kneeland, Igor Kabiljo, Dmitry Shchukin, Andre Rodrigues, Scott Michelson, Ben Christensen, Kaushik Veeraraghavan, and Chunqiang Tang. RAS: Continuously Optimized Region-Wide Datacenter Resource Allocation. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles*, 2021.

[37] NVIDIA Multi-Instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[38] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013.

[39] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[40] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 1–18. USENIX Association, July 2021.

[41] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. Efficient Queue Management for Cluster Scheduling. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–15, 2016.

[42] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, et al. Presto: SQL on Everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1802–1813. IEEE, 2019.

[43] Dharma Shukla, Muthian Sivathanu, Srinidhi Viswanatha, Bhargav Gulavani, Rimma Nehme, Amey Agrawal, Chen Chen, Nipun Kwatra, Ramachandran Ramjee, Pankaj Sharma, Atul Katiyar, Vipul Modi, Vaibhav Sharma, Abhishek Singh, Shreshth Singhal, Kaustubh Welankar, Lu Xun, Ravi Anupindi, Karthik Elangovan, Hasibur Rahman, Zhou Lin, Rahul Seetharaman, Cheng Xu, Eddie Ailijiang, Suresh Krishnappa, and Mark Russinovich. Singularity: Planet-Scale, Preemptive and Elastic Scheduling of AI Workloads. *arXiv preprint arXiv:2202.07848*, 2022.

[44] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. Twine: A Unified Cluster Management System for Shared Infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 787–803. USENIX Association, 2020.

[45] Zhenheng Tang, Shaohuai Shi, Xiaowen Chu, Wei Wang, and Bo Li. Communication-Efficient Distributed Deep Learning: A Comprehensive Survey. *arXiv preprint arXiv:2003.06307*, 2020.

[46] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. *Advances in neural information processing systems*, 30, 2017.

[47] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[48] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th ACM European Conference on Computer Systems*, 2015.

[49] Shaoqi Wang, Oscar J. Gonzalez, Xiaobo Zhou, Thomas Williams, Brian D. Friedman, Martin Havemann, and Thomas Woo. An Efficient and Non-Intrusive GPU Scheduling Framework for Deep Learning Training Systems. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2020.

[50] Zhijun Wang, Huiyang Li, Zhongwei Li, Xiaocui Sun, Jia Rao, Hao Che, and Hong Jiang. Pigeon: an Effective Distributed, Hierarchical Datacenter Job Scheduler. In *Proceedings of the ACM symposium on cloud computing*, pages 246–258, 2019.

[51] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages

945–960, Renton, WA, April 2022. USENIX Association.

[52] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, Carlsbad, CA, October 2018. USENIX Association.

[53] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548. USENIX Association, November 2020.

[54] Peifeng Yu and Mosharaf Chowdhury. Fine-Grained GPU Sharing Primitives for Deep Learning Applications. In *Proceedings of Machine Learning and Systems*, volume 2, pages 98–111, 2020.

[55] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. *HotCloud*, 10(10-10):95, 2010.

[56] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. Fuxi: A Fault-Tolerant Resource Management and Job Scheduling System at Internet Scale. *Proc. VLDB Endow.*, 7(13):1393–1404, aug 2014.

[57] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis C.M. Lau, Yuqi Wang, Yifan Xiong, and Bin Wang. HiveD: Sharing a GPU Cluster for Deep Learning with Guarantees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 515–532. USENIX Association, November 2020.

[58] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Kevin Wilfong, Sundaram Narayanan, Jack Langman, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. Understanding Data Storage and Ingestion for Large-Scale Deep Recommendation Model Training: Industrial Product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 1042–1057, 2022.

[59] Ningxin Zheng, Huiqiang Jiang, Quanlu Zhang, Zhenhua Han, Lingxiao Ma, Yuqing Yang, Fan Yang, Chengruidong Zhang, Lili Qiu, Mao Yang, and Lidong Zhou. PIT: Optimization of Dynamic Sparse Deep Learning Models via Permutation Invariant Transformation. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 331–347, New York, NY, USA, 2023. Association for Computing Machinery.