Understanding Data Storage and Ingestion for Large-Scale Deep Recommendation Model Training

Industrial Product*

Mark Zhao†, Niket Agarwal†, Aarti Basant†, Buğra Gedik†, Satadru Pan†, Mustafa Ozdal†, Rakesh Komuravelli†, Jerry Pan†, Tianshu Bao†, Haowei Lu†, Sundaram Narayanan†, Jack Langman†, Kevin Wilfong†, Harsha Rastogi†, Carole-Jean Wu†, Christos Kozyrakis‡, Parik Pol†

†Meta, ‡Stanford University

ABSTRACT

Datacenter-scale AI training clusters consisting of thousands of domain-specific accelerators (DSA) are used to train increasingly-complex deep learning models. These clusters rely on a data storage and ingestion (DSI) pipeline, responsible for storing exabytes of training data and serving it at tens of terabytes per second. As DSAs continue to push training efficiency and throughput, the DSI pipeline is becoming the dominating factor that constrains the overall training performance and capacity. Innovations that improve the efficiency and performance of DSI systems and hardware are urgent, demanding a deep understanding of DSI characteristics and infrastructure at scale.

This paper presents Meta's end-to-end DSI pipeline, composed of a central data warehouse built on distributed storage and a Data PreProcessing Service that scales to eliminate data stalls. We characterize how hundreds of models are collaboratively trained across geo-distributed datacenters via diverse and continuous training jobs. These training jobs read and heavily filter massive and evolving datasets, resulting in popular features and samples used across training jobs. We measure the intense network, memory, and compute resources required by each training job to preprocess samples during training. Finally, we synthesize key takeaways based on our production infrastructure characterization. These include identifying hardware bottlenecks, discussing opportunities for heterogeneous DSI hardware, motivating research in datacenter scheduling and benchmark datasets, and assimilating lessons learned in optimizing DSI infrastructure.

CCS CONCEPTS

• **Software and its engineering** → **Distributed systems organizing principles**; • **Information systems** → *Database management system engines*; • **Computing methodologies** → **Machine learning**.

*This paper is part of the Industry Track of ISCA 2022's program.

ISCA '22, June 18–22, 2022, New York, NY, USA

ACM ISBN 978-1-4503-8610-4/22/06…$15.00
https://doi.org/10.1145/3470496.3533044

KEYWORDS

Machine learning systems, databases, distributed systems, data ingestion, data storage

ACM Reference Format:
Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, Parik Pol. 2022. Understanding Data Storage and Ingestion for Large-Scale Deep Recommendation Model Training: Industrial Product. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3470496.3533044

1 INTRODUCTION

Domain-specific accelerators (DSAs) for deep neural networks (DNNs) have become ubiquitous because of their superior performance per watt over traditional general purpose processors [40]. Industry has rapidly embraced DSAs for both DNN training and inference. These DSAs include both traditional technologies, such as GPUs and FPGAs, as well as application-specific integrated circuits (ASICs) from, e.g., Habana [37], Graphcore [45], SambaNova [67], Tenstorrent [74], Tesla [75], AWS [23], Google [40], and others.

DSAs are increasingly deployed in immense scale-out systems to train increasingly-complex and computationally-demanding DNNs using massive datasets. For example, the latest MLPerf Training round (v1.1) [56] contains submissions from Azure and NVIDIA using 2048 and 4320 A100 GPUs, respectively, whereas Google submitted training results using pods containing up to 4096 TPUv4s [58]. At Meta, we are building *datacenter-scale AI training clusters* by both scaling our production datacenters to include thousands of GPUs using ZionEX nodes [59] and building the AI Research SuperCluster (RSC) with over ten thousand GPUs [10]. These DSAs have been laser-focused on optimizing and scaling compute for training, namely matrix-heavy computations used during backpropagation.

In reality, DNN training in production involves significantly more than just backpropagation. Hazelwood *et al.* noted how *"For many machine learning models at [Meta], success is predicated on the availability of extensive, high-quality data"* [38]. Namely, a **data storage and ingestion** (DSI) pipeline, consisting of *offline data generation, dataset storage, and online preprocessing services*, must store and feed exabytes of data to high-performance training nodes (trainers). The design of the DSI pipeline significantly affects the overall DNN training capacity and performance, but has received little consideration compared to model training itself.

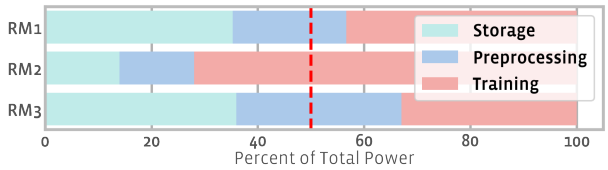

Figure 1: Percent of storage, preprocessing, and training power required to train three production DLRMs, with line drawn at 50%. DLRMs exhibit diverse DSI resource requirements and can consume more power than training.

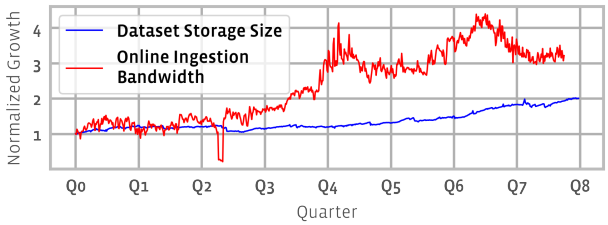

Figure 2: Normalized training dataset size and online data ingestion bandwidth across our recommendation models. Storage and bandwidth is in the exabytes and hundreds of Tbps and has grown by over 2x and 4x over the past two years, respectively.

This paper focuses on *understanding DSI requirements, unique workload characteristics, and systems for industry-scale, deep learning recommendation model (DLRM) training*. We focus on DLRMs because DLRMs a) underpin many of Meta's personalization and ranking services [18, 35, 36, 38, 54], b) consume the vast majority of the overall ML training cycles [18] (and DSI capacity) in Meta datacenters, and c) introduce novel DSI challenges not yet captured by current ML benchmarks [53] nor considered by existing systems [41, 47, 49, 57, 61, 87, 88].

Understanding and efficiently scaling the DSI pipeline is essential in enabling large-scale training for several reasons. First, inefficiencies in the pipeline cripple training throughput [57], underutilizing expensive DSAs. Second, DSI infrastructure competes for valuable power resources with trainers. Figure 1 shows how *storage and online preprocessing can already consume more power than the actual GPU trainers themselves in Meta's datacenters.* This directly constrains training capacity due to fixed datacenter power budgets [24]. Finally, steady innovation in model complexity and DSAs for training are increasing data storage and bandwidth demands. Figure 2 shows how industry-scale dataset sizes and online data ingestion bandwidth requirements have grown by over 2× and 4× over the past two years, respectively. Barring similar innovation for DSI, we expect DSI infrastructures to severely limit training capacity at the datacenter-scale as training DSAs continue to yield higher performance per watt.

In this paper, we present Meta's end-to-end DSI pipeline, which enables large-scale ML model training at scale. Training data is generated by extract-transform-load (ETL) jobs that transform unstructured feature data and event logs collected across production fleets into structured training samples. Petabyte-scale datasets are held in a centralized data warehouse as Hive tables [76], which are subsequently stored as optimized columnar files in Tectonic [64], Meta's append-only distributed filesystem. Finally, to handle intense online preprocessing demands, we present a production-deployed disaggregated online preprocessing framework called *Data PreProcessing Service (DPP)* that iteratively reads and transforms mini-batches of training data, scaling from 10s to 100s of preprocessing nodes for each training job.

A key contribution of this paper lies in the deep system performance characterization for Meta's production-deployed DSI pipeline, supporting large-scale DNN training. We describe Meta's collaborative feature engineering and model training process for DLRMs, which reveals key system design requirements and optimizations for Meta's underlying global DSI infrastructure. Features for DLRM training evolve rapidly in our datasets, and samples are constantly generated. This requires us to store and serve *massive and dynamically-changing feature sets*, representing exabytes of cumulative storage. Each training job requires an *online preprocessing step, demanding significant compute, network, and memory resources*, in order to extract, transform, and load samples into materialized tensors for training.

Table 1 summarizes the design principles of our DSI pipeline presented in Section 3, and key takeaways and open research problems for the wider community that we present in Section 7. It connects each to specific system characterization results we present throughout Sections 4 to 6 for the production-deployed DSI pipeline.

In summary, our primary contributions are:

- We describe and identify the DSI pipeline as a critical and equally important, yet vastly understudied, component of datacenter-scale ML training infrastructures.
- We provide an end-to-end description and share the design rationales behind our production-deployed DSI pipeline architecture, tailor-made to meet important requirements of DLRM training at scale.
- We perform a deep characterization of industry-scale DLRM training workloads (summarized in Table 1), including coordinated training, data generation and storage, and online preprocessing, on our production hardware — identifying critical bottlenecks and key insights.
- We provide our outlook of important and open research questions for systems and computer architects to design and scale the DSI pipeline for large-scale training.

We hope that this work distills meaningful architecture and systems challenges in ML, beyond just DSAs for DNN training, and will guide the community to identify and focus on DL workloads that are representative of industry uses.

2 RECOMMENDATION MODEL BACKGROUND

Personalized recommendation models are used to suggest new, relevant content to users to provide meaningful interactions. These models are highly potent across a breadth of tasks. They leverage features from a user and a potential recommendation, and output a prediction (e.g., click-through rate) of how likely the user is to interact with the recommendation. For example, a video hosting site may use a user's set of liked videos and candidate videos' genres to rank new videos to recommend to the user.

Recommendation models are trained using mini-batch stochastic gradient descent (SGD) [69], similar to most vision and natural

Table 1: Summary of key takeaways and research opportunities, motivated by characterization results.

Key Lessons Learned and Takeaways	Related Characterization Results (§4-§6)
§3: Data storage and ingestion must be disaggregated to meet the large dataset capacity and online preprocessing requirements of training jobs.	§4: Diverse training jobs run continuously in geo-distributed datacenters. §5: Datasets exceed local storage capacity and are heavily filtered during reading. §6: Using trainer CPUs for expensive online preprocessing results in data stalls.
§7.1: There are significant compute, network, memory, and disk bottlenecks across storage, preprocessing, and trainer node hardware. These hardware bottlenecks are key areas of optimization for DSI.	§5.1: Heavy feature filtering requires high IOPS from storage nodes. §6.2: Data loading at trainers requires considerable front-end trainer host resources. §6.3: Data extraction and transformation will be increasingly constrained by memory bandwidth.
§7.2: Using and designing heterogeneous hardware for dataset storage systems and data ingestion can address many hardware bottlenecks, but require overcoming key challenges that arise in DSI pipelines.	§5.2: Training jobs reuse popular features which can be cached in high IOPS/W storage. §6.2: Data loading requires expensive "datacenter tax" operations. §6.4: We identified common transforms, especially feature generation, that are highly-resource intensive on CPUs.
§7.3: Designing datacenters for ML training require carefully provisioning compute, network, and storage capacity based on training jobs' diverse DSI requirements. Intelligent global training job schedulers can further improve efficiency across geo-replicated datacenters.	§4.1: Industry-scale training requires a collaborative training process across many models. §4.2: Each model requires periodic combo jobs with high concurrent compute demands that must be co-located with storage and scheduled across global regions.
§7.4: Benchmarks are vital for guiding research directions. Current public datasets are not representative of industry; there is need for research in developing benchmark datasets. We identify important characteristics of industrial datasets.	§4.3: Industrial datasets are constantly updated with new data and features. §5.1: They are PB-sized and stored as structured samples in distributed file systems. §5.1: Training jobs read one epoch with feature-wise and row-wise filtering.
§7.5: DSI power constrains training capacity — systems efficiency optimizations are vital. Efficiency optimizations must be co-designed across hardware/software and the end-to-end DSI pipeline. We walk through recent optimization examples.	Optimizations must consider application characteristics (selective reading - §5.1, feature popularity - §5.2), hardware performance and bottlenecks (HDD IOPS - §5.1, preprocessing memory bandwidth - §6.3), and trade-offs across end-to-end DSI pipeline systems (§3).

language processing (NLP) models. SGD generalizes a model to complex distributions by iteratively updating the model's weights to minimize a loss function, given successive *mini-batches* of samples. Each training sample is represented as a preprocessed tensor of *features* and a label.

Our production recommendation models are built on the open-source DLRM architecture [63]. Modern DLRMs are massive, consisting of over 12 trillion parameters to train, requiring $\approx$ 1 zetaFLOPs of total compute [59]. To meet the compute requirements of DLRM training, we built the ZionEX hardware platform. Each node contains 8 NVIDIA A100 GPUs connected with NVLink for intra-node communication. Each GPU has a dedicated 200 Gbps RDMA over Converged Ethernet (RoCE) NIC for inter-node communication over a dedicated backend network, connecting thousands of GPUs to form a datacenter-scale AI training cluster. A node also contains four CPU sockets, each with a dedicated 100 Gbps NIC connected to our regular datacenter network for data ingestion. We defer readers to [59] for more details on ZionEX.

To fully leverage our training hardware, we use data [46] and model [30] parallelism by replicating and sharding the model across multiple ZionEX training nodes. We distribute different mini-batches to each trainer. Trainer nodes synchronize embeddings, activations, and gradients with each other using collective communication primitives over the backend network, iterating until a certain model quality metric (e.g., normalized entropy [39]) is reached. At Meta, we use hundreds of distinct recommendation models in production across our services. We continuously train new versions of each model offline (Section 4), and update production models periodically [38].

Each training job relies on a data storage and ingestion (DSI) pipeline to supply each trainer with training data throughout the duration of the job. The DSI pipeline is thus responsible for *generating* training samples, *storing* them into datasets, and *preprocessing* samples into tensors loaded in device memory, i.e., GPU HBMs.

3 DISAGGREGATED DATA STORAGE, INGESTION, AND TRAINING PIPELINE

In this section, we present our DSI requirements and the storage, preprocessing, and training systems that compose Meta's DSI pipeline as shown in Figure 3.

3.1 Data Generation and Storage

Overview and Requirements. Figure 3 shows how fresh training samples are continuously generated by our model serving framework in order to ensure model accuracy and comply with privacy requirements [38]. Each sample is created as services evaluate a user and item using the model serving framework. The framework first generates an extensive set of *features* (e.g., a user's liked pages) as input to an appropriate model, which outputs a prediction used for recommendation tasks. The requesting service then monitors *events* representing the outcome of each recommendation (e.g., if a user interacted with a post). These features and events are logged at serving time to avoid data leakage [44] between model serving and training. Subsequent streaming and batch extract-load-transform (ETL) jobs continuously join and label raw feature and event logs into labeled and schematized samples.

Training samples are placed in a storage solution that must meet several key requirements. First, individual tables require *tens of thousands of features, with features being constantly added or removed.* Training jobs must be able to *dynamically and selectively read* stored features. Second, *developer productivity is paramount.* We must allow ranking engineers and the underlying infrastructure to easily work across hundreds of models and tables via a centralized data warehouse using a common schema. Furthermore, ranking engineers frequently run interactive queries using Spark [84] or Presto [71] as a part of feature engineering in addition to training. Finally, *datasets are continuously updated with fresh samples.*

3.1.1 Data Generation. To extract and aggregate billions of features and events across the entire fleet each day, we rely on Scribe [43] — Meta's global distributed messaging system. Each service responsible for serving models or handling interactions continuously passes raw feature and event logs to a Scribe daemon running on

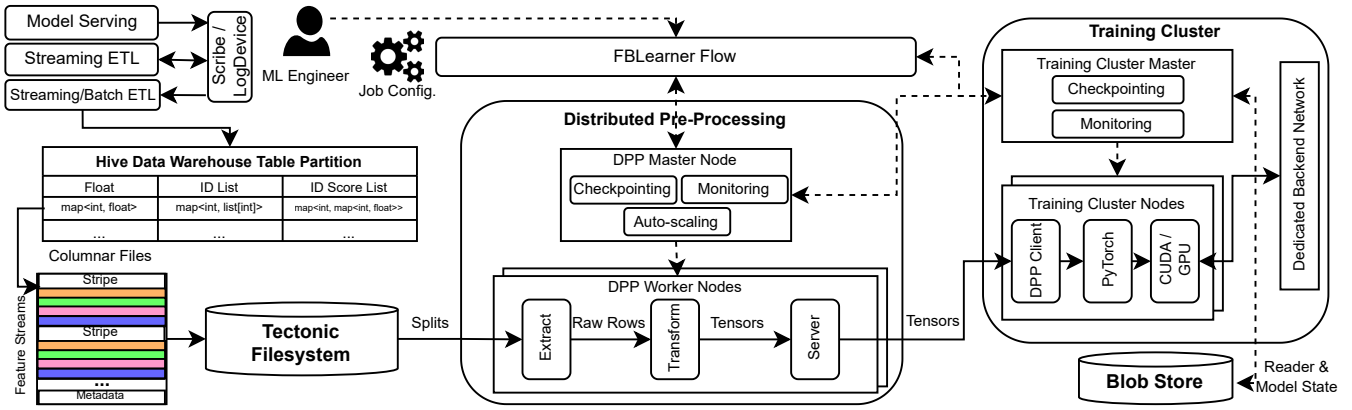

Figure 3: Production data storage and ingestion pipeline architecture. Solid and dashed lines represent data and control flow, respectively.

every host. Scribe then groups logs into record-oriented logical streams and stores each stream into LogDevice [52] — a reliable distributed store for append-only, trimmable streams built on top of RocksDB [13].

To update production models, streaming engines first join and label raw feature and event logs from Scribe and publish labeled samples into various Scribe streams used to update in-production models. Various streaming and batch processing engines, such as Spark [84], further join, label, and filter samples from Scribe streams to produce partitioned (e.g., hourly or daily) offline datasets used to train new production model versions. Because traditional engines work well to generate training data, require comparatively little compute and power resources to dataset storage and ingestion, and are not on the critical path of training, we elide in-depth discussion here.

3.1.2 Data Storage. We store training samples in a data warehouse as partitioned Hive [76] tables because of Hive's compatibility with both internal systems and open source engines including Spark and Presto. Samples are represented as structured rows, each containing features and labels, with features requiring the vast majority ($> 99\%$) of stored bytes. To ensure interoperability of the DSI pipeline across hundreds of models and tens of thousands of features, we store two types of features, dense and sparse, in map columns. A dense feature column maps a feature ID to a continuous value (e.g., current time). A sparse feature column maps a feature ID to a *variable length* list of categorical values (e.g., page IDs). Some sparse features are stored in an additional column that further associates each categorical value with a floating point "score" used for weighing (e.g., page creation time).

We encode Hive tables in a columnar file format (DWRF), forked from Apache ORC [4]. Like ORC, rows are stored across multiple files. Each file contains a set of stripes, representing a number of table rows. Stripes are further divided into compressed and encrypted streams. A key distinction of DWRF is the ability to enable feature filtering at the storage layer by flattening each feature column and storing features as thousands of separate logical columns at the file layer (see Section 7). Each flattened feature column is subsequently encoded as one or more streams, depending on its schema and encoding. Stripes are periodically flushed and appended to the file. Files are written in Tectonic [64] — Meta's exabyte-scale distributed append-only filesystem. Tectonic splits files into durable blocks distributed across HDD storage nodes.

3.2 Online Preprocessing

Overview and Requirements. Each training job uses an online (training-time) preprocessing pipeline to continuously transform raw samples in storage into *preprocessed* tensors in a trainer's memory. Like offline data generation, online preprocessing is commonly subdivided into ETL phases.

Raw bytes are *extracted* from storage and decoded into training samples, a process involving filtering, decryption, decompression, reconstruction, and other format transformations. Training samples are next *transformed* into tensors. Float values may be normalized, and categorical values may be hashed, clipped, or even sorted. New features may even be derived from multiple raw features. Once features are preprocessed, they are batched together into tensors. The tensors are *loaded* into trainers, usually in device memory (e.g., HBM of GPUs).

Online preprocessing has distinct requirements that differ from those met by traditional ETL engines. First, transformations for online preprocessing are localized to each mini-batch, not across many rows. Second, online preprocessing is on the critical path of training and must match throughput required by trainers. Right-sizing online preprocessing throughput is critical to avoid either over-provisioning resources or introducing data stalls [57] that will bottleneck and under-utilize expensive trainers. Finally, online preprocessing runs concurrently alongside *each* training job, requiring significant cumulative compute and power resources that scales with training capacity, unlike offline data generation.

3.2.1 Scalable Preprocessing with DPP. *Data PreProcessing Service* (DPP) is our disaggregated service that provides online preprocessing for training jobs across the datacenter fleet. DPP is responsible for reading raw training data from storage, preprocessing it into ready-to-load tensors, and supplying the tensors to each training node's PyTorch [65] runtime. We designed DPP to both *scale to right-sized resources and eliminate data stalls across disparate jobs*, as well as *enable vital productivity mechanisms for developers and ML engineers*. We meet these requirements by dividing DPP into a data and control plane, which are designed to enable application

throughput and ease-of-use, respectively. The control plane consists of a DPP Master, and the data plane consists of DPP Workers and Clients. As shown in Figure 3, ML engineers launch training jobs via FBLearner Flow [32], which then launches a DPP Master and at least one DPP Worker on general-purpose compute nodes.

DPP Control Plane. At the beginning of each training job, the DPP Master receives a session specification (a PyTorch DataSet) that reflects the preprocessing workload, containing the dataset table, specific partitions, required features, and transformation operations for each feature. The DPP Master enables scalable work distribution by breaking down the entire preprocessing workload, across petabytes of data, into independent and self-contained work items for the data plane called *splits* that represent successive rows of the entire dataset. The Master serves splits to DPP Workers upon request and tracks progress as splits are completed.

In addition to work distribution, the DPP Master is responsible for fault tolerance and auto-scaling. The DPP Master periodically creates a checkpoint which can be used to restore reader state on failure. The DPP Master also continuously monitors Worker health, automatically restarting any Workers that have failed without needing a checkpoint restore due to Workers' stateless design. The DPP Master itself is replicated to avoid being a single point of failure.

Finally, the DPP Master implements auto-scaling via a controller. The controller collects utilization (CPU, memory, and network) statistics and the number of buffered tensors from each DPP Worker. It then periodically evaluates scaling decisions, calculating the number of DPP Workers to either drain or launch with the goal of maintaining a non-zero number of buffered tensors (indicating that trainer demand is met) and maximum CPU, network, and memory utilization. In doing so, the DPP Master eliminates data stalls with minimal DPP resource requirements.

DPP Data Plane. DPP Workers and Clients are responsible for data plane operations of DPP. Workers are designed to effortlessly scale out to eliminate data stalls. Workers are stateless, precluding any limit to how many Workers can exist in a given DPP session. They only communicate with the DPP Master (to fetch work items) and a limited number of DPP Clients (to serve tensors); all transformations within a mini-batch are performed locally. On startup, each Worker pulls a set of transformations from the Master, represented by a serialized and compiled PyTorch module, that it will use during preprocessing. Workers then continuously query for and process splits from the DPP Master.

As shown in Figure 3, each split requires Workers to extract, transform, and (partially) load training data. Specifically, Workers begin by reading, decompressing, and decrypting raw Tectonic chunks. Sets of raw chunks are then reconstructed into streams and decoded into raw table rows, filtering out unused features if necessary. Next, it applies the specified transformations to each raw feature using high-performance C++ binaries. Once features are transformed, Workers batch samples together into tensors to be loaded onto GPU trainers. We ensure that transient delays in the pipeline do not introduce data stalls by maintaining a small buffer of tensors in each Worker's memory.

Trainers. DPP Clients form the other half of the data plane. A Client runs on each training node, exposing a hook that the PyTorch runtime can call to obtain preprocessed tensors. These requests are transparently transformed into a simple RPC request which returns a batch of tensors from the Worker buffer. By offloading computationally expensive operations to DPP Workers and enabling Client multithreading, Clients do not bottleneck the data ingestion pipeline. To ensure that Client and Worker network connections can scale, each Client uses partitioned round robin routing, capping the number of connections that Clients and Workers need to maintain.

To enable large-scale recommendation model training, we have built high-performance training clusters [59], enabling individual jobs to train on hundreds to thousands of GPUs. Each training job is controlled by a Trainer Master, which manages the overall training session. On each trainer, a PyTorch runtime manages the local training workflow, transferring preprocessed tensors between the DPP Client and GPU device memory. Parameter updates between trainers occur over a dedicated backend network and do not impact data ingestion.

4 COORDINATED TRAINING AT SCALE

Next, we explore how DLRMs are trained and deployed at Meta. We do so because large-scale training deployments require thousands of training jobs over hundreds of models and datasets, all running on a shared global infrastructure. Understanding industrial training jobs highlights important system and infrastructure implications not found in commonly-studied hyperparameter (HP) tuning or isolated training jobs. We focus on the three representative recommendation models (RMs) highlighted in Figure 1, denoted $RM_{1,2,3}$, as these models are the most widely-used and training resource-intensive.

4.1 Collaborative Release Process

Hundreds of recommendation models are deployed in production at Meta, each continuously developed by many training jobs. Each training job produces a new model version with the goal of becoming the next production model. Many of our models are supported by large teams of ranking engineers, requiring the need to allow continuous experimentation across engineers while avoiding conflicts between model versions and conserving limited training capacity. This need naturally arises as model engineering teams mature [66].

We thus adopted a regimented release process which occurs over three phases. First, ML engineers *explore* their ideas (e.g., new features or model architectural improvements) on top of the current production model through hundreds to thousands of small training jobs. Exploratory jobs generally require less compute and use a small fraction (typically $< 5\%$) of its respective table's total samples. Next, the most promising ideas are periodically *combined* in various permutations to generate tens to hundreds of training jobs. These *combo jobs* are large and trained within a short window, demanding immense parallelism and the majority of the table. Finally, the most promising *release candidates* (RCs) are further trained and evaluated on fresh data, and the most accurate model is deployed in production. While these jobs are large, there are only a few.

Counter-intuitively, the model release process can result in *more* diversity in terms of temporal locality, model architectures, and feature sets than in isolated or HP tuning jobs [41, 47]. This is because training capacity is highly-constrained compared to per-job compute requirements, requiring engineers to combine many architecture and feature proposals into one combo job. Figure 4 illustrates how the model design space is explored given compute constraints.

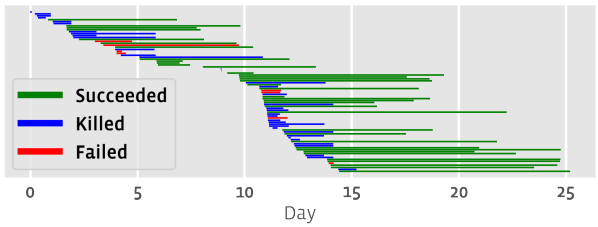

Figure 4: Chart showing skewed and variable training duration and status of 82 RM_1 combo jobs within one model release iteration.

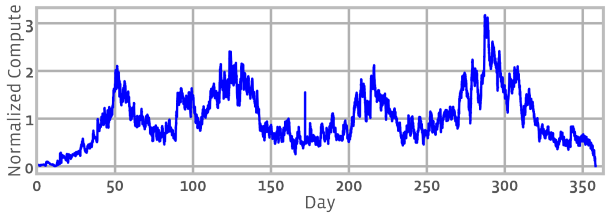

Figure 5: Normalized daily peak compute utilization over all collaborative training jobs over one year, showing peaks corresponding to combo jobs.

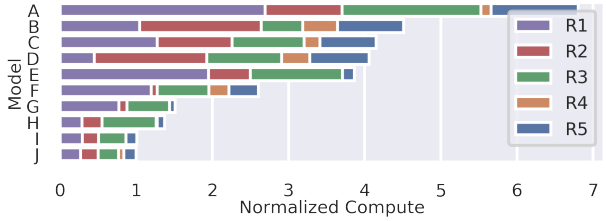

Figure 6: Compute demand over ten most commonly-used models (A-J), split by global region (R1-R5), normalized to model J.

While individual jobs are long-running and can take over 10 days to train, many jobs fail or are killed because their performance is lackluster. Instead of waiting to launch jobs synchronously, engineers will immediately schedule new jobs to maximize the number of explored ideas within the combo time window, resulting in a large temporal skew between jobs.

4.2 Global Training Utilization

The aforementioned collaborative training jobs, including exploratory, combo, and release candidates, run on the global fleet of training (including DSI) infrastructure, spread across global regions, each with multiple datacenters [38]. Figure 5 shows a historical utilization, in terms of normalized compute, of all collaborative training jobs across DLRMs over one calendar year across our entire fleet. We observe distinct peaks in utilization, corresponding to periods where many models concurrently train combo jobs. Because these combo jobs are on the critical path of model release, we must explicitly architect our datacenters with sufficient storage, preprocessing, and training capacity to meet the peak utilization of combo jobs.

Furthermore, as we explore in Section 5, each model reads a distinct dataset. At the same time, cross-region (and often cross-datacenter) bandwidth is highly-constrained. This requires systems and datacenter architects to co-locate DSI resources with trainers themselves and provision enough capacity for each. Figure 6 shows a bar chart of the relative compute demand of the ten most commonly-run models, broken down by the region in which they ran. Our global scheduler currently balances training jobs for each model across regions, requiring each region to contain a copy of all models' datasets. Bin-packing opportunities can reduce storage costs, with care to ensure data availability for each model as its peak compute demand can exceed regional capacity.

Table 2: Number of features created for RM_1 dataset within a 6 month window and their status 6 months later.

Beta	Experimental	Active	Deprecated	Total
10148	883	1650	1933	14614

4.3 Feature Engineering

The sets of features stored to a dataset and read by training jobs can also vary heavily, as features also undergo rapid experimentation and productionization. To understand feature storage variability, Table 2 shows the total number of new features proposed for RM_1's production dataset within a 6-month window and the status of the feature 6 months later. Beta features are not actively logged, but may be back-filled or injected (i.e., dynamically joined) for each exploratory training job. Experimental features are used as a part of combo or release candidate jobs. If the release candidate job becomes the next production model, its used features become active, while some older features may become deprecated following a review process or even reaped to protect user privacy. Experimental, active, and deprecated features are actively written to the dataset. We observe that features are rapidly changing in production datasets, with hundreds of new features added and deprecated each month. Thus, efficient ML data storage infrastructure must adapt to frequent changes in the feature set.

4.4 Summary of Key Takeaways

Training production models requires a collaborative release process across hundreds of engineers. Critically, ideas are periodically amalgamated in a large number of concurrent combo jobs for each model, resulting in large peaks in training and DSI resources across our fleet during this phase. Because combo jobs are on the critical path of model release, we must design datacenters with sufficient capacity across global regions for peak demand corresponding to combo jobs. This capacity required is spread across hundreds of models with varying compute demand, motivating the need for efficient co-location and scheduling of jobs and datasets across regions to reduce inter-region storage and network demands. Finally, we explored how training jobs are temporally skewed and exhibit diverse model architectures, and datasets are continuously evolving, inhibiting system optimizations that assume highly-synchronized and similar training jobs or static datasets, e.g. [41, 47, 57].

5 UNDERSTANDING DATA STORAGE AND READING

We next explore how datasets are stored in our data warehouse and read by training jobs, highlighting implications to our storage hardware and infrastructure.

Table 3: Compressed sizes of all table partitions, each partition, and the cumulative partitions used by a representative release candidate training job for each RM.

Model	All Partitions (PB)	Each Partition (PB)	Used Partitions (PB)
RM1	13.45	0.15	11.95
RM2	29.18	0.32	25.94
RM3	2.93	0.07	1.95

Table 4: Feature characteristics of production models.

Model Class	# Dense Features	# Sparse Features	# Derived Features
RM1	1221	298	304
RM2	1113	306	317
RM3	504	42	1

Table 5: Dataset characteristics for each model.

Dataset	# Float Feats.	# Sparse Feats.	Avg. Coverage	Avg. Sparse Feat. Length	% Feats. Used	% Bytes Used
RM1	12115	1763	0.45	25.97	11	37
RM2	12596	1817	0.41	25.57	10	34
RM3	5707	188	0.29	19.64	9	21

5.1 Individual Jobs Read and Filter Large Datasets

Benchmark datasets are typically re-read multiple times (epochs) to reach target accuracy [53]. Thus, existing work focuses on randomly modifying [19, 26–28, 49] or caching [47, 57, 61, 82] data across epochs to improve DSI efficiency.

Unlike these benchmarks, production training jobs are not constrained by the amount of data. Instead, model size and compute capacity limits constrain the number of features and samples each job can use, respectively. Production training jobs do not require stochastic preprocessing across multiple epochs, but instead can reach a desired target accuracy with (less than) one epoch containing many samples. Individual training jobs specify a given table as its dataset, along with filters that select a subset of data within the table along two dimensions: a variable number of partitions (row filter) and a set of features within each sample to read (column filter).

We begin by analyzing one representative RC training job from $RM_{1,2,3}$. Table 3 shows the (compressed) size characteristics of each model's respective production training data table. It also shows the size of each partition in the table (partitioned by date) and the cumulative size of the partitions used by the training job for each *RM*. Even our largest training jobs often read less than the entire available dataset and each sample only once. The partitions that are read still require petabytes of data, which is significantly larger than the local storage capacity at each trainer node, contrary to prior assumptions [57]. Furthermore, as shown in Figure 2, dataset sizes for production models are continuously growing, driven by multiple factors such as organic user growth, reduced downsampling, and an increase in engineered features.

Next, we study how a training job selects data along the feature (column) dimension. Individual training jobs specify a *feature projection*, consisting of a list of desired features to be read from all rows in the designated partitions. Table 4 shows the number of dense, sparse, and derived features required by a representative RC model version for each *RM*. These model versions require

Table 6: I/O Sizes for features read by an RM1 training job.

	Mean	Std	p5	p25	p50	p75	p95
I/O Size (B)	23.2K	117K	18	451	1.24K	3.92K	97.7K

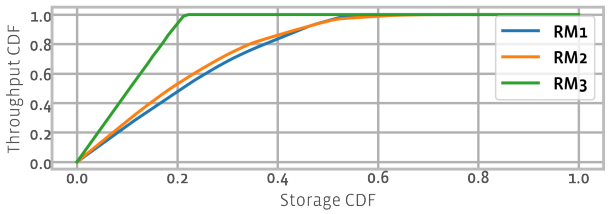

Figure 7: CDF of popular bytes to throughput absorbed, across one month of each RM's runs. Popular bytes are reused across runs.

504 – 1221 and 42 – 306 dense and sparse features, respectively. This is in contrast to Table 5, which shows that significantly more features are logged in each model's table. Each training job only needs to read 9 – 11% of stored features. Even when accounting for the number of bytes read, Table 5 highlights that $RM_{1,2,3}$ only read between 21 and 37 percent of stored bytes across used partitions. The relative increase in read bytes is because read features typically exhibit larger coverage (i.e., fraction of samples logging the feature) and sparse feature lengths, and thus require more bytes, as these features contribute stronger signals to model quality and are thus favored by ML engineers.

While there appears to be room to reduce feature collection, feature experimentation is essential for ML engineers. We prioritize developer productivity and heavily err on the side of keeping features, even at the cost of storage, to ensure that ML engineers have access to the features they need.

Selective reading also has further implications for the performance of our storage nodes. Table 6 shows the distribution, in bytes, of a representative RM_1 training job's I/O sizes from storage. Heavy filtering and columnar storage of features on disk (Section 3) results in relatively-small contiguous regions for read features. Further software-hardware co-design is needed to ensure that disk seeks do not cripple storage IOPS.

5.2 Data is Reused Across Training Jobs

While individual training jobs require extensive filtering, training jobs do collectively reuse data. Inter-job data reuse can occur throughout the model release process because ML engineers do not develop an entirely new model architecture and feature set each iteration, but instead largely build upon a common baseline (e.g., the current production model version).

Figure 7 shows that, based on training runs for $RM_{1,2,3}$ over one month, training runs for each model tend to favor specific bytes. The x-axis shows a CDF of bytes within the model's used set of table partitions. The y-axis shows the percent of all I/O from storage that the most-popular x percent of stored bytes contribute to. To serve 80% of traffic from storage, we only require the most commonly-used 39, 37, and 18 percent of RM_1's, RM_2's, and RM_3's datasets, respectively. Combined with Table 5, Figure 7 also highlights how used features and bytes vary across training jobs. RM3 exhibits little variance in features — Table 5 and Figure 7 show that both individual and collective models read roughly 21% of the stored

Table 7: Insufficient GPU trainer host resources for preprocessing introduces significant data stalls for RM_1.

% of GPU Stall Time	% CPU Utilization	% Memory BW Utilization
56	92	54

Table 8: RMs drive large and diverse throughput at each GPU training node.

	RM1	RM2	RM3
GPU Trainer Throughput (GB/s, per 8-GPU Node)	16.50	4.69	12.00

bytes. Meanwhile, RM_1 and RM_2 show high variance; individual jobs only read 37% and 34% of the stored bytes, respectively, while jobs collectively read over 60% of the stored bytes.

5.3 Summary of Key Takeaways

In this section, we explored how features are stored in petabyte-scale datasets that greatly exceed local storage capacities, requiring training jobs to read samples from a centralized data warehouse (Section 3.1). Furthermore, each training job requires extensive filtering both in the number of samples (rows) and features extracted from each sample (columns). Column-wise filtering results in small reads from storage because features are stored in columnar files. Finally, across training jobs for a given model, we observed significant reuse in commonly-used features, with 40% of bytes contributing to over 80% of read throughput.

6 UNDERSTANDING ONLINE PREPROCESSING

Distributed trainers drive strict data ingestion bandwidth requirements. Data stalls result when online preprocessing throughput is less than the aggregate throughput of the trainers themselves, underutilizing GPU resources [57].

Current preprocessing solutions, which perform preprocessing on the CPUs of each training node, can cause data stalls. To demonstrate this, we ran a training job for RM_1 on a training node consisting of two 28-core x86 CPU sockets, two 100 Gbps frontend NICs for data ingestion, and a total of 8 NVIDIA V100 GPUs. The trainer read from distributed storage, preprocessed each mini-batch using the production PyTorch [65] stack, and performed training on the same machine. Table 7 shows that 56% of GPU cycles were spent stalled waiting for training data. The high CPU utilization shows that the trainer's CPUs cannot preprocess data fast enough to serve the GPUs, motivating us to build DPP to eliminate data stalls (Section 3.2.1).

We next seek to understand the preprocessing bottlenecks of our production DSI pipeline in detail. To do so, we analyzed the preprocessing throughput demanded by GPUs across RMs. We then traced through data loading resource requirements at GPU trainers, and data extraction and transformation resource requirements at DPP Workers.

6.1 GPU Training Throughput

We measured the online preprocessing throughput required by each *RM* by running a production training job for each *RM* on a training node, ensuring that GPUs were not stalled by serving tensors from an in-memory buffer. Table 8 shows the per-trainer node GPU throughput requirements (i.e., tensor ingestion rate) for each representative *RM*. GPU throughput requirements are not only significant, but vary by over 6× across models. The difference in throughput across the models is due to the variations in operational intensity (i.e., compute per sample) across models, as well as synchronization overheads between GPUs during each iteration.

Furthermore, we project the online preprocessing throughput requirement to increase by 3.5× within the next two years due to larger training samples, improved hardware accelerators, and software optimizations. We cannot simply over-provision resources for preprocessing at each trainer for the worst-case model; doing so would waste large amounts of capacity and power across our fleet. The DSI pipeline must instead scale online preprocessing resources to meet intense and increasing GPU throughput demands, and adapt to the diverse requirements across models.

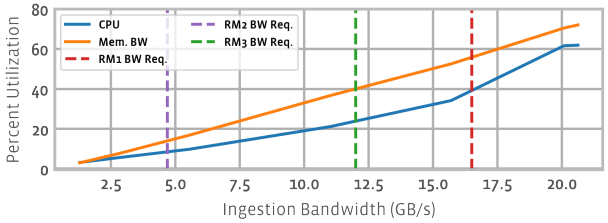

Figure 8: CPU and memory bandwidth utilization at trainer frontend as data loading throughput scales. Vertical lines show the network utilization of each RM.

6.2 Data Loading at GPU Trainers

Section 3 described how our DSI pipeline allows training jobs to scale online preprocessing resources by loading preprocessed tensors from distributed DPP Workers. We now show that data loading over the network at GPU trainers, even without extraction or transformations, still requires significant trainer CPU, network, and memory bandwidth.

Figure 8 shows the memory bandwidth and CPU utilization on the 2-socket, 8-GPU training node as we increase the rate at which preprocessed tensors are loaded from a set of DPP Workers. Vertical lines represent the required GPU throughput across each *RM*, as measured in Table 8. High training data throughput demands driven by the GPUs directly translate to considerable front-end resource requirements for data loading. Production-scale model training is approaching NIC saturation, even with significant reduction in data sizes due to transformation operations (see Section 6.3). Second, even without expensive extraction or transformation operations, production models require up to 40% of CPU cycles (almost a full socket) and 55% of memory bandwidth to load training data. This demand is due to network stack and memory management requirements in addition to the necessary "datacenter tax" [42] operations such as TLS decryption and Thrift deserialization that are required in our production environment. Considering memory bandwidth saturates at $\approx$ 70% utilization, data loading constrains trainers' compute, memory bandwidth, and network resources.

Table 9: DPP Worker throughput across RMs and the resulting # Workers required to meet trainer demands. Storage RX is compressed and transform RX/TX is uncompressed.

Model	kQPS	Storage RX (GB/s)	Transform RX (GB/s)	Transform TX (GB/s)	# DPP Workers Req. per GPU Training Node
RM1	11.623	0.8	1.37	0.68	24.16
RM2	7.995	1.2	0.96	0.50	9.44
RM3	36.921	0.8	1.01	0.22	55.22

6.3 Extracting & Transforming Data at DPP Workers

Data extraction and transformation requires strikingly more resources than are available on trainers and must be distributed.

Table 9 shows the maximum data extraction and transformation throughput achieved by each DPP Worker, running on a general purpose server (C-v1, Table 10). We need large and highly-variable throughput across models to meet the GPU demands in Table 8 — between 9 and 55 servers per trainer node. Unlike traditional distributed query executors [55, 60, 83, 84], which rely on large clusters to produce a result as fast as possible, online preprocessing requires continuous throughput guided by GPU demands; allocating more preprocessing workers will not improve end-to-end training time. On the other hand, using trainer hosts for preprocessing is also insufficient, as our online preprocessing demands represent considerably more network, compute, and memory bandwidth resources than available locally, especially when factoring in data loading. Not only do models require large and diverse data loading throughput, achievable extract and transform throughput, given fixed compute, varies across models, emphasizing the need to right-size preprocessing resources to each model.

To understand the implications of these results in more detail, we observe that preprocessing significantly reduces data sizes, especially considering storage bytes are compressed in Table 9. This is due to a combination of filtering, over reading features from storage (see Section 7), and size reduction during transformations. This has implications on network throughput requirements, as 1.18 to 3.64× more network bandwidth is required to extract raw samples from storage than to load preprocessed tensors. Thus, performing data extraction at the trainers would further amplify the network bandwidth requirements beyond the per-model requirements shown in Figure 8, resulting in data stalls. Table 9 shows how DPP Workers are bound on ingress NIC bandwidth for RM_2, requiring $\approx$ 10 Gbps of our current 12.5 Gbps NICs (Table 10), reaching practical NIC throughput limits. Even given higher NIC bandwidth limits, the datacenter network can pose potential bottlenecks, as upper datacenter network links are often oversubscribed [24, 34, 68, 72].

Furthermore, network bandwidth is not the only limiting factor. Figure 9 shows DPP CPU and memory bandwidth utilization at saturation. Each model exhibits diverse resource requirements. RM_1 is bottlenecked on memory bandwidth and CPU utilization. This is because RM_1 requires significantly more CPU cycles for preprocessing due to its computationally expensive transformations. RM_3 is bound on memory capacity, forcing us to limit the worker thread pool size to avoid OOM exceptions.

While models are constrained on various hardware currently, these bottlenecks will change as future generations of compute

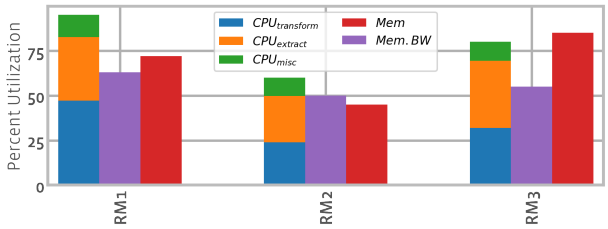

Figure 9: CPU, memory, and memory bandwidth utilization at preprocessing workers across RMs. CPU utilization is broken down into transformation, extraction, and miscellaneous cycles.

Table 10: Hardware specifications across three versions of compute servers, plus a hypothetical state-of-the-art server.

Node	Num. Physical Cores	NIC (Gbps)	Memory (GB)	Peak Mem. BW (GB/s)	Peak Mem. BW / Core (GB/s)	NIC BW / Core (Gbps)
C-v1	18	12.5	64	75	4.2	0.69
C-v2	26	25.0	64	92	3.5	0.96
C-v3	36	25.0	64	83	2.3	0.69
C-vSotA	64	100.0	1024	205	3.2	1.56

Table 11: Description of common preprocessing transformations.

Op Name	Description
Cartesian	Compute Cartesian product between two sparse features
Bucketize	Shard features based on bucket borders
ComputeScore	Arithmetic operations on sparse features
Enumerate	Similar to Python enumerate()
PositiveModulus	Compute positive modulus on sparse features
IdListTransform	Performs intersection of two sparse feature lists
BoxCox	BoxCox transform for normalization
Logit	Logit transform for normalization
MapId	Maps feature IDs to fixed values
FirstX	List truncation of sparse features for normalization
GetLocalHour	Compute local timestamp
SigridHash	Compute hash value to normalize list of sparse features
NGram	Compute an n-gram between multiple sparse features
Onehot	Apply one hot encoding to normalize dense features
Clamp	Same as std::clamp
Sampling	Randomly sample training data samples

nodes provide a different balance of compute, memory, and network. Table 10 shows the hardware characteristics of our current (C-v1) and upcoming versions of compute nodes used by DPP. To discount for the multifaceted and potentially non-technical factors impact the final specification of future compute nodes, we also show a hypothetical compute node (C-vSotA) based on state-of-the-art hardware (e.g., AMD Milan with 8 channels of DDR4-3200 [11], and a 100G NIC). We observe an increased scaling in the number of cores and NIC bandwidth compared to memory bandwidth; we thus expect memory bandwidth to become the bottleneck as per-core memory bandwidth decreases compared to per-core NIC bandwidth. To demonstrate this, we ran preprocessing for RM_2 on the C-v2 node in Table 10 and observed that memory bandwidth, not network, was the bottleneck. We further studied where memory bandwidth is going, and observed that 50.4%, 24.9%, 16.4%, and 4.7% of LLC misses were due to transformations, extraction, network receive, and network send, respectively, highlighting areas for future optimization.

6.4 Transformation Operations

Accelerating transformation operations (e.g., via GPUs [1]) are promising, but requires more research as our transformations can be distinctly different from the matrix-heavy operations used in training and may contend cycles with training. Table 11 provides a list of important (but not exhaustive) preprocessing transformations that are needed by our production DLRMs. These operations are distinctly different from the image-centric operations used in many preprocessing libraries [6], such as crops, resizing, and color augmentations.

DLRM transformation operations can be split into three classes: feature generation, sparse feature normalization, and dense feature normalization. Dense feature normalization (Logit, BoxCox, Onehot) and sparse feature normalization (SigridHash, FirstX) normalize features based on dataset statistics. Feature generation operations (commonly-used ones include Bucketize, NGram, and MapId) derive new dense and sparse features from raw dataset features. Feature generation is especially expensive — dense normalization, sparse normalization, and feature generation typically require around 5%, 20%, and 75% of transformation cycles, respectively. We are actively open-sourcing these operations in TorchArrow [15].

6.5 Summary of Key Takeaways

DLRM training jobs require online preprocessing that can induce data stalls on GPU trainers due to limited host compute, memory, and network resources. We built DPP, a disaggregated online preprocessing service, to completely eliminate data stalls by offloading data extraction and transformation operations. Nevertheless, trainers must be provisioned with enough host resources to handle data loading rates driven by the GPUs. At disaggregated DPP Workers, we expect memory bandwidth to become the primary bottleneck, largely due to transformations. Finally, we identified how DLRM transformation operations differ from image models, and feature generation dominates transformation compute.

7 KEY INSIGHTS AND RESEARCH OPPORTUNITIES

This section assimilates and explores key insights we learned while architecting and profiling Meta's DSI pipeline and important research challenges we continue to face. As DSAs for ML continue to improve, the DSI pipeline is becoming an increasingly resource-intensive component of the end-to-end ML training pipeline. We argue that similar attention to DSI is warranted in order to continue scaling datacenter-scale training.

7.1 Hardware Bottlenecks in DSI

Storage layer. Cross-datacenter bandwidth is constrained (Section 4.2). Thus, we must provision sufficient storage capacity and IOPS bandwidth within our production storage layer (i.e., Tectonic [64]) in each datacenter. Storage capacity requirements are driven by the industry-scale dataset sizes (Table 3). IOPS bandwidth requirements are driven by the overall trainer node throughput (Table 8) and scaled by data volume changes due to preprocessing. We must also factor in how I/O size characteristics (Table 6) affect achievable IOPS of the HDD storage nodes. As it stands, we observe an over 8× throughput-to-storage gap even after accounting for triplicate replication for durability. In order to meet IOPS demands, we must provision significantly more storage capacity per datacenter than is required to store datasets, motivating the need for storage hardware and systems that better balance IOPS and storage capacity.

Data ingestion layer. Section 6.3 highlighted how the diverse set of preprocessing requirements across RMs constrained compute, network, and memory resources at DPP Workers. However, Table 10 shows the expected growth of network bandwidth and compute capacity will outpace memory bandwidth for the next generations of our general-purpose compute nodes. We expect data ingestion to be heavily constrained by memory bandwidth, warranting further research in methods to reduce memory bandwidth demand during preprocessing.

Trainers. The goal of the DSI pipeline is to avoid data stalls, ensuring that trainers (and specifically DSAs) are fully utilized and dictate the throughput demands of the end-to-end training pipeline. Section 6 explains how without DPP, trainers are constrained on both front-end network and host memory and compute resources due to online preprocessing. After disaggregating DPP, we can achieve our goal of ensuring that the accelerators are fully fed by simply provisioning enough host compute, memory bandwidth, network resources for data loading when designing our training nodes. For example, our next-generation ZionEX nodes contain 4 CPU sockets, each with a 100 Gbps dedicated front-end NIC, to ensure that all GPUs are fully fed [59].

7.2 Heterogeneous Hardware for DSI

The bottlenecks above allude to ample opportunity to leverage heterogeneous hardware across data storage and ingestion.

Balancing storage throughput and capacity. We noted how our HDD-based storage nodes presented ≈ 8× throughput-to-storage gap, requiring us to provision datacenters with excess storage capacity to meet IOPS demand. We can improve the power efficiency of our storage layer by balancing storage throughput and capacity by leveraging heterogeneous storage media with a higher IOPS per watt. For example, our SSD-based storage nodes can provide 326% IOPS per watt, but trades off storage capacity with only 9% capacity per watt, compared to HDDs within our fleet. At the same time, simply placing training data on SSDs would result in an unfavorable storage-to-throughput gap due to our large datasets.

Storage layers for DSI should balance storage throughput and capacity for optimal power efficiency. There are further software and hardware optimization opportunities, such as placing commonly-used features (Figure 7) on SSD-based caches, or leveraging non-volatile memory. These solutions must consider important characteristics of industry-scale DSI pipelines characterized in Sections 4 and 5. Datacenter architects must balance storage and IOPS capacity in each datacenter while reasoning about highly-variable training demand, shared capacity across multiple datasets, and dynamic scheduling behavior across geo-distributed datacenters. The storage layer must also accurately predict and place commonly-used bytes on the appropriate medium, requiring complex predictions on locality of highly asynchronous training jobs and the reuse patterns of continuously-evolving feature sets and model architectures.

Accelerating transformations. We also foresee further opportunities for acceleration during online preprocessing. For example, recent efforts have been used to accelerate preprocessing on the training GPU [1], but risks degrading training throughput. While we believe preprocessing can be accelerated, there are numerous challenges to address.

Transformations can be performed on the GPU trainer, host CPU, disaggregated CPUs, or disaggregated accelerators — deciding the optimal placement is non-trivial. Many transformation operations described in Section 6 exhibit diverse amenability for acceleration; the most efficient hardware platform can vary across operations. For example, we observed an 11.9× and 1.3× GPU/CPU performance for SigridHash and Bucketize, respectively, on a V100 GPU and 20 CPU threads. The prevalence of each operation, and thus the most efficient preprocessing solution, also varies heavily across models.

In fact, each model requires a large graph of many operations across its feature set. For example, a single feature X may require a DAG of multiple operations that apply *Bucketize* to feature A, apply *FirstX* to feature B, compute the *Ngram* of the intermediate values, and apply *SigridHash* to generate feature X. Current GPU hardware and APIs are optimized for large, parallel singular tasks as opposed to multiple small, diverse operations. Launching a kernel for each feature incurs significant launch and host-to-device transfer overheads. For example, we observed over three orders of magnitude throughput speedup by applying a simple kernel on a tensor combining 1000 sparse features versus applying the same kernel on each feature separately. Furthermore, transformation operations produce outputs and intermediate values with variable-length sequences, requiring complex memory management not well addressed by current GPUs and ML frameworks [33]. Addressing these characteristics of preprocessing is critical in order to fully leverage the acceleration potential of GPUs, FPGAs, or other DSAs.

Accelerating data extraction and loading. Accelerators can be applied beyond transformation operations. Section 6 showed how datacenter tax operations further constrain DPP Worker resources. Necessary TLS operations amplify memory bandwidth by 3×, further constraining the limited bandwidth resource. Hardware such as accelerators for microservices [73] and SmartNICs that support techniques such as TLS offloading will be needed to further scale DSI.

7.3 Datacenter Planning and Scheduling

We must intelligently design and provision compute, network, and storage capacity in each datacenter to ensure both high utilization of each DSI system given a fixed power budget and sufficient capacity to meet peak training demands. To do so, we rely on extensive models built by continuously profiling DSI workloads on both storage nodes and DPP Workers. Designing a datacenter for ML training thus requires not only understanding compute requirements of the models, but also an accurate benchmark of the datasets and preprocessing requirements (which we discuss next).

We must also consider scheduling as an important component of ML training systems. We have multiple datacenters in a region and multiple regions globally. Section 4 explored how our training workload is spread across regions, requiring datasets to be replicated and co-located with trainers. We foresee opportunity for a global scheduler to intelligently route and bin-pack training jobs to specific regions to reduce storage duplication. Furthermore, as our datasets continue to grow beyond DC-scale, other parallelism techniques, such as model-hopper parallelism [62] to move TB-scale models across DCs instead of PB-scale data, will become increasingly important.

7.4 Representative Benchmark Datasets

Recent advancements in DSAs have largely been driven and measured by benchmarks. For example, the latest round of MLPerf Training [53] (v1.1) saw results from 14 organizations with up to 2.3× improvement over the previous round. Unfortunately, these benchmarks have largely focused on models as opposed to datasets, leading to rapid innovation in model architectures and training nodes while largely ignoring the DSI pipeline. Current leading benchmark datasets, such as ImageNet [31] and COCO [50], were released in 2010 and 2014 and have been largely unchanged, despite their ubiquity in academia. The Criteo 1TB Click Logs [8], the Recommendation dataset used by MLPerf Training [80], was released in 2013.

Section 5 describes how production recommendation datasets differ from static benchmark datasets. Understanding how representative datasets are generated, stored, and read are critical to further research in DSI for ML training. Specifically, we characterized that for DLRMs, training samples are a) continuously generated from real-world events, b) stored as structured samples in a common data warehouse or feature store which are further stored as columnar files in a distributed filesystem, and c) require petabytes of storage. Furthermore, when training jobs read the dataset, they d) perform extensive online preprocessing operations, e) only require one epoch, and f) require further row-wise and feature-wise filtering. We are actively working towards more representative benchmark datasets (e.g., [7]), and we hope this paper motivates the importance of further work in this area.

7.5 Multi-dimensional System Co-design

Efficiency gains solely via hardware are slow, as hardware refreshes must be planned years in advance. To continuously improve DSI efficiency (and increase training capacity), *we spend significant effort optimizing our DSI systems because at scale, small efficiency gains can translate to MWs of additional trainer capacity.* In our experience, DSI system architects must understand and co-design optimizations across two dimensions. **Top-to-bottom:** DSI systems must leverage characteristics and meet requirements of applications (ML models and engineers) while optimizing for the underlying hardware. **End-to-end:** Optimizations must be considered across the DSI components; optimizations can trade-off efficiency across the entire pipeline. We highlight this through recent examples next.

Feature flattening. Our warehouse tables' schema, which store features in maps, are designed to handle constantly-evolving feature sets. This required training jobs to read the entire row, resulting in a large "over read" of bytes from storage as each training job only requires a small set of features from each row. To maintain the usability benefits (i.e., rapid feature engineering) of our map schema for applications while avoiding over reads, we optimized our DWRF file format with *feature flattening*. As shown in Figure 10,

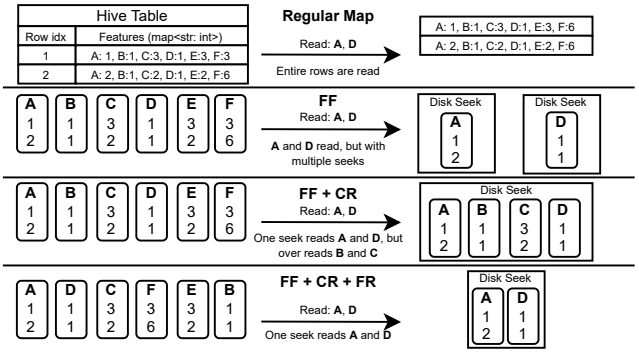

Figure 10: Example highlighting which features are read given regular map, FF=feature flattening, CR=coalesced reads, and FR=feature reordering.

Table 12: Normalized DPP Worker and storage throughput as a result of progressive optimizations. FF = Feature Flattening, FM = In-Memory Flatmap, LO = Localized Optimizations, CR = Coalesced Reads, FR = Feature Reordering, LS = Large Stripes.

	Baseline	+FF	+FM	+LO	+CR	+FR	+LS
DPP Throughput	1.00	2.00	2.30	2.94	2.94	2.94	2.94
Storage Throughput	1.00	0.03	0.03	0.03	0.99	1.84	2.41

feature flattening organizes maps such that values for a given map key (feature ID), across rows, are stored as separate streams on disk. Combined with additional metadata for each feature ID in the DWRF file, readers can selectively read the desired features from storage. Table 12 shows how feature flattening doubled DPP Worker throughput (with 12% increase in storage capacity) due to a reduction in CPU cycles spent extracting unnecessary data.

Coalesced reads. Unfortunately, filtering at the storage layer reduced I/O sizes from almost 8 MB (Tectonic's chunk size) to the small I/O sizes ($\approx$ 20 KB) shown in Table 6, resulting in poor IOPS on our HDD-based storage nodes due to excessive disk seeks. This reduced storage throughput by 97%, as shown in Table 12. We optimized top-to-bottom for our HDDs by *coalescing reads*. Figure 10 shows how coalesced reads group selected feature streams within 1.25 MiB in one I/O, eliminating storage throughput degradation by amortizing disk seeks.

Feature reordering. Coalesced reads resulted in over reads of unused features, limiting its effectiveness. These over reads occur because the offline data generation step effectively orders feature streams randomly. In the example in Figure 10, reading features (A, D) with a coalesced read ends up over reading (B, C), squeezed between (A, D). We applied end-to-end optimization by augmenting our data generation path to continuously write feature streams in each file ordered based on features' popularity in training jobs launched within a recent window (e.g., 7 days). Feature reordering leverages the insight that a small set of popular features contribute to a large percent of storage throughput (Section 5.2), reducing the amount of unnecessary features in a coalesced read and improving storage throughput by 84%.

Large stripes, in-memory flatmaps, and localized optimizations. Finally, we applied numerous other optimizations across the DSI pipeline. We used *large stripes* that increase the number of rows in each file stripe to increase the average I/O size of each read. Increasing stripes to $\approx$ 1 GB further improved storage throughput by 31%. We also noted how both DWRF and tensor formats represent each feature's values contiguously across rows, while data extraction reconstructs all features in a row-based map format, requiring costly format changes and copies between columnar and row formats during preprocessing. We changed how DPP Workers represent samples with *in-memory flatmaps* to match the DWRF and tensor formats. This reduced format conversions and thus reduced constrained memory bandwidth demands, improving Worker throughput 15%. Finally, even localized optimizations at DPP Workers, such as removing unnecessary null checks and using build-time optimizations like Linker Time Optimization (LTO) and AutoFDO [25], further improved DPP throughput by 28%.

The optimizations above required us to consider DSI optimizations top-to-bottom and end-to-end. For example, combining feature flattening, coalesced reads, and feature reordering considered ML application characteristics and demands, such as filtering on evolving feature sets, while addressing seek overheads of our underlying HDD hardware (top-to-bottom optimization). Similarly, we demonstrated how feature flattening trades-off storage capacity for DPP efficiency, as well as how optimizations such as large stripes, feature reordering, and in-memory flatmaps are made across the end-to-end DSI pipeline from dataset generation to online preprocessing.

In total, Table 12 shows how these optimizations increased DPP and storage throughput by 2.94× and 2.41×, respectively. When weighed by our provisioned DPP and storage power requirements, these co-designed optimizations resulted in a 2.59× reduction in DSI power requirements, allowing us to provision datacenters with significantly more compute resources for trainers.

We are continuing to explore promising co-designed optimization opportunities that aim at improving data extraction and optimizing DPP in-memory formats, such as Velox [16] and TorchArrow [15]. We are also exploring other optimization techniques, such as caching preprocessed tensors and balancing transformations between offline and online ETL.

8 RELATED WORK

Data storage and ingestion for ML. We presented Meta's production deployed DSI pipeline.

ETL pipelines. We discussed how we use traditional ETL engines, such as Spark [84], to generate structured training data from raw logs. A number of query and streaming engines [21, 55, 60, 83–85] are used across industry for this task. We characterized how online data preprocessing demands diverge from traditional ETL, requiring deep integration into PyTorch, pipelined compute, localized minibatch transforms, and right-sizing, highlighting a need for a distinct online preprocessing framework that optimizes for power efficiency while eliminating data stalls.

Data storage and warehousing for ML. We characterized how industrial datasets differ from benchmarks, requiring massive and evolving datasets, highly selective filtering, and interoperability and reuse across multiple models and systems. To address these needs, we store datasets as Hive [76] tables on top of Tectonic [64] using an optimized Apache ORC [4] like format. Comparable solutions

exist across industry. Feature stores (e.g., Tecton [9]) and data warehouses (e.g., DeltaLake [22] and Snowflake [29]) manage datasets. These rely on variety of storage and memory formats [2, 3, 5, 14].

Online preprocessing for ML. tf.data [61] presents a runtime and API for online preprocessing in TensorFlow [17]. While tf.data Service [12] is an experimental feature to distribute online preprocessing on a user-managed cluster, tf.data focuses on optimizing preprocessing on the host CPU. DPP similarly presents a runtime for online preprocessing fully integrated in PyTorch [65]. DPP is inherently disaggregated and runs as a fully-managed service at Meta, enabling online preprocessing to automatically scale to meet the throughput required by training jobs running on hundreds of GPUs.

Other recent works target key DSI components, focusing on benchmark vision and NLP models. CoorDL [57], Quiver [47], and DIESEL [77] are caches that optimize for single-server training, HP tuning jobs, and small files, respectively. DeepIO [87] and DLFS [88] leverage hardware (RDMA and NVMeOF) to provide randomized minibatches from storage. Revamper [49] randomly augments samples across epochs to reduce online preprocessing costs. Wang *et al.* [78] and Kumar *et al.* [48] mitigate data stalls on TPUs. OneAccess [41] motivates sharing online preprocessing for HP tuning jobs. Industrial DSI characteristics are markedly different from benchmarks (Section 7), limiting the impact of such systems in industrial settings.

Understanding Large-Scale Training. tf.data [61] characterized online preprocessing at Google, highlighting similar findings such as prevalent data reuse and demanding compute requirements. Xin *et al.* [81] characterized how ML models are trained and deployed at Google, focusing on model lifecycle management. Pulkit *et al.* [20] presented MLdp, a data management platform at Apple, focusing on industrial data lifecycle management requirements including versioning, provenance, and access control. Hazelwood *et al.* [38] presented how Meta trains and serves ML models at scale. To the best of our knowledge, our work represents the first characterization of the end-to-end DSI workloads, systems, and infrastructure in a large-scale training deployment, noting key research opportunities.

Recommendation Models. Gupta *et al.* [36] analyzed industry-scale inference models on three different CPU architectures. DeepRecSys [35] and RecSSD [79] optimized inference requests across CPUs and GPUs, and SSDs, respectively. Acun *et al.* [18] characterized DLRM architectures on GPU trainers, Sethi *et al.* [70] presented an optimized embedding sharding strategy for DLRM training, Maeng *et al.* [51] explored checkpointing trainer state, and AIBox [86] optimized training using hierarchical memory for parameters. However, these prior works do not discuss the DSI pipeline, a critical part of ML training.

9 CONCLUSION

DSI infrastructure will dominate large-scale training resource and power capacity without further innovation and optimization. This paper presented Meta's end-to-end data storage, ingestion, and training pipeline used to train our production recommendation models. We characterized DSI workloads on our fleet, including coordinated training, dataset storage and reading, and online preprocessing workloads. To spur further research, we synthesized key insights and important research directions gleaned from our characterization and experience.

ACKNOWLEDGMENTS

We would like to thank the many engineers in the numerous infrastructure and hardware teams that build, support, and maintain the systems and hardware that compose Meta's DSI pipeline. We also thank Daniel Ford, Dheevatsa Mudigere, Chunqiang Tang, Matei Zaharia, and the anonymous reviewers for their feedback on this paper. Christos Kozyrakis is supported by the Stanford Platform Lab and its affiliate members.

REFERENCES

[1] 2021. NVIDIA Data Loading Library (DALI). https://developer.nvidia.com/dali
[2] 2022. Apache Arrow. https://arrow.apache.org/
[3] 2022. Apache Avro. https://avro.apache.org/
[4] 2022. Apache ORC. https://orc.apache.org/
[5] 2022. Apache Parquet. https://parquet.apache.org/
[6] 2022. DALI Supported Operations. https://docs.nvidia.com/deeplearning/dali/user-guide/docs/supported_ops.html
[7] 2022. Datasets for the Deep Learning Recommendation Model (DLRM). https://github.com/facebookresearch/dlrm_datasets
[8] 2022. Download Criteo 1TB click Logs dataset. https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/
[9] 2022. Enterprise feature store for machine learning. https://www.tecton.ai/
[10] 2022. Introducing the AI research SuperCluster - Meta's cutting-edge AI supercomputer for AI Research. https://ai.facebook.com/blog/ai-rsc/
[11] 2022. Milan - Cores - AMD. https://en.wikichip.org/wiki/amd/cores/milan
[12] 2022. Module: Tf.data.experimental.service : Tensorflow core v2.6.0. https://www.tensorflow.org/api_docs/python/tf/data/experimental/service
[13] 2022. A persistent key-value store. http://rocksdb.org/
[14] 2022. TFRecord. https://www.tensorflow.org/tutorials/load_data/tfrecord
[15] 2022. TorchArrow. https://github.com/facebookresearch/torcharrow
[16] 2022. Velox. https://github.com/facebookincubator/velox
[17] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi
[18] B. Acun, M. Murphy, X. Wang, J. Nie, C. Wu, and K. Hazelwood. 2021. Understanding Training Efficiency of Deep Learning Recommendation Models at Scale. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 802–814. https://doi.org/10.1109/HPCA51647.2021.00072
[19] Naman Agarwal, Rohan Anil, Tomer Koren, Kunal Talwar, and Cyril Zhang. 2020. Stochastic Optimization with Laggard Data Pipelines. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 10282–10293. https://proceedings.neurips.cc/paper/2020/file/74dbd1111727a31a2b825d615d80b2e7-Paper.pdf
[20] Pulkit Agrawal, Rajat Arya, Aanchal Bindal, Sandeep Bhatia, Anupriya Gagneja, Joseph Godlewski, Yucheng Low, Timothy Muss, Mudit Manu Paliwal, Sethu Raman, Vishrut Shah, Bochao Shen, Laura Sugden, Kaiyu Zhao, and Ming-Chuan Wu. 2019. Data Platform for Machine Learning. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1803–1816. https://doi.org/10.1145/3299869.3314050
[21] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, out-of-Order Data Processing. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1792–1803. https://doi.org/10.14778/2824032.2824076
[22] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał undefinedwitakowski, Michał Szafrański, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage

over Cloud Object Stores. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3411–3424. https://doi.org/10.14778/3415478.3415560

[23] AWS. 2022. AWS EC2 Trn1 Instances. https://aws.amazon.com/ec2/instance-types/trn1/

[24] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. 2018. The datacenter as a computer: Designing warehouse-scale machines. *Synthesis Lectures on Computer Architecture* 13, 3 (2018), i–189.

[25] Dehao Chen, Tipp Moseley, and David Xinliang Li. 2016. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 12–23.

[26] Dami Choi, Alexandre Passos, Christopher J. Shallue, and George E. Dahl. 2020. Faster Neural Network Training with Data Echoing. arXiv:1907.05550 [cs.LG]

[27] E. D. Cubuk, B. Zoph, D. Mane, V. Vasudevan, and Q. V. Le. 2019. AutoAugment: Learning Augmentation Strategies From Data. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 113–123. https://doi.org/10.1109/CVPR.2019.00020

[28] Ekin Dogus Cubuk, Barret Zoph, Jon Shlens, and Quoc Le. 2020. RandAugment: Practical Automated Data Augmentation with a Reduced Search Space. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 18613–18624. https://proceedings.neurips.cc/paper/2020/file/d85b63ef0ccb114d0a3bb7b7d808028f-Paper.pdf

[29] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 215–226. https://doi.org/10.1145/2882903.2903741

[30] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. 2012. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf

[31] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[32] Jeffrey Dunn. 2018. Introducing FBLearner Flow: Facebook's AI backbone. https://engineering.fb.com/2016/05/09/core-data/introducing-fblearner-flow-facebook-s-ai-backbone/

[33] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. 2021. TurboTransformers: An Efficient GPU Serving System For Transformer Models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 389–402. https://doi.org/10.1145/3437801.3441578

[34] Nathan Farrington and Alexey Andreyev. 2013. Facebook's data center network architecture. In *2013 Optical Interconnects Conference*. Citeseer, 49–50.

[35] U. Gupta, S. Hsia, V. Saraph, X. Wang, B. Reagen, G. Wei, H. S. Lee, D. Brooks, and C. Wu. 2020. DeepRecSys: A System for Optimizing End-To-End At-Scale Neural Recommendation Inference. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 982–995. https://doi.org/10.1109/ISCA45697.2020.00084

[36] U. Gupta, C. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang. 2020. The Architectural Implications of Facebook's DNN-Based Personalized Recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 488–501. https://doi.org/10.1109/HPCA47549.2020.00047

[37] Habana. 2022. Habana Homepage. https://habana.ai/

[38] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 620–629. https://doi.org/10.1109/HPCA.2018.00059

[39] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, and Joaquin Quiñonero Candela. 2014. Practical Lessons from Predicting Clicks on Ads at Facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising* (New York, NY, USA) *(ADKDD'14)*. Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/2648584.2648589

[40] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

[41] Aarati Kakaraparthy, Abhay Venkatesh, Amar Phanishayee, and Shivaram Venkataraman. 2019. The Case for Unifying Data Loading in Machine Learning Clusters. In *USENIX HotCloud*. https://www.microsoft.com/en-us/research/publication/the-case-for-unifying-data-loading-in-machine-learning-clusters/

[42] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-Scale Computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 158–169. https://doi.org/10.1145/2749469.2750392

[43] Manolis Karpathiotakis, Dino Wernli, and Milos Stojanovic. 2019. Scribe: Transporting petabytes per hour via a distributed, Buffered queueing system. https://engineering.fb.com/2019/10/07/data-infrastructure/scribe/

[44] Shachar Kaufman, Saharon Rosset, Claudia Perlich, and Ori Stitelman. 2012. Leakage in Data Mining: Formulation, Detection, and Avoidance. *ACM Trans. Knowl. Discov. Data* 6, 4, Article 15 (Dec. 2012), 21 pages. https://doi.org/10.1145/2382577.2382579

[45] Simon Knowles. 2021. Graphcore. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. 1–25. https://doi.org/10.1109/HCS52781.2021.9567075

[46] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. arXiv:1404.5997 [cs.NE]

[47] Abhishek Vijaya Kumar and Muthian Sivathanu. 2020. Quiver: An Informed Storage Cache for Deep Learning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 283–296. https://www.usenix.org/conference/fast20/presentation/kumar

[48] Sameer Kumar, James Bradbury, Cliff Young, Yu Emma Wang, Anselm Levskaya, Blake Hechtman, Dehao Chen, HyoukJoong Lee, Mehmet Deveci, Naveen Kumar, Pankaj Kanwar, Shibo Wang, Skye Wanderman-Milne, Steve Lacy, Tao Wang, Tayo Oguntebi, Yazhou Zu, Yuanzhong Xu, and Andy Swing. 2021. Exploring the limits of Concurrency in ML Training on Google TPUs. arXiv:2011.03641 [cs.LG]

[49] Gyewon Lee, Irene Lee, Hyeonmin Ha, Kyunggeun Lee, Hwarim Hyun, Ahnjae Shin, and Byung-Gon Chun. 2021. Refurbish Your Training Data: Reusing Partially Augmented Samples for Faster Deep Neural Network Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 537–550. https://www.usenix.org/conference/atc21/presentation/lee

[50] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. 2015. Microsoft COCO: Common Objects in Context. arXiv:1405.0312 [cs.CV]

[51] Kiwan Maeng, Shivam Bharuka, Isabel Gao, Mark Jeffrey, Vikram Saraph, Bor-Yiing Su, Caroline Trippel, Jiyan Yang, Mike Rabbat, Brandon Lucia, and Carole-Jean Wu. 2021. Understanding and Improving Failure Tolerant Training for Deep Learning Recommendation with Partial Recovery. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 637–651. https://proceedings.mlsys.org/paper/2021/file/b73ce398c39f506af761d2277d853a92-Paper.pdf

[52] Mark Marchukov. 2017. LogDevice: A distributed data store for logs. https://engineering.fb.com/2017/08/31/core-data/logdevice-a-distributed-data-store-for-logs/

[53] Peter Mattson, Christine Cheng, Gregory Diamos, Cody Coleman, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debo Dutta, Udit Gupta, Kim Hazelwood, Andy Hock, Xinyuan Huang, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St John, Carole-Jean Wu, Lingjie Xu, Cliff Young, and Matei Zaharia. 2020. MLPerf Training Benchmark. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 336–349. https://proceedings.mlsys.org/paper/2020/file/02522a2b2726fb0a03bb19f2d8d9524d-Paper.pdf

[54] Ivan Medvedev, Haotian Wu, and Taylor Gordon. 2019. Powered by AI: Instagram's Explore recommender system. https://ai.facebook.com/blog/powered-by-ai-instagrams-explore-recommender-system/

[55] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 330–339. https://doi.org/10.14778/1920841.1920886

[56] MLCommons. 2021. MLPerf Training v1.1 Results. https://mlcommons.org/en/training-normal-11/

[57] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. 2021. Analyzing and Mitigating Data Stalls in DNN Training. In *Proceedings of the VLDB Endowment*. https://www.microsoft.com/en-us/research/publication/analyzing-and-mitigating-data-stalls-in-dnn-training/

[58] Samuel Moore. 2021. *Here's How Google's TPU v4 AI Chip Stacked Up in Training Tests.* https://spectrum.ieee.org/heres-how-googles-tpu-v4-ai-chip-stacked-up-in-training-tests

[59] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie Amy Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. 2022. Software-Hardware Co-design for Fast and Scalable Training of Deep Learning Recommendation Models. In *2022 ACM/IEEE 49th Annual International Symposium on Computer Architecture (ISCA)*.

[60] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[61] Derek G. Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. 2021. tf.data: A Machine Learning Data Processing Framework. In *Proceedings of the VLDB Endowment*. https://doi.org/10.14778/3476311.3476374

[62] Supun Nakandala, Yuhao Zhang, and Arun Kumar. 2020. Cerebro: A Data System for Optimized Deep Learning Model Selection. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2159–2173. https://doi.org/10.14778/3407790.3407816

[63] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* abs/1906.00091 (2019). https://arxiv.org/abs/1906.00091

[64] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, JR Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. 2021. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 217–231. https://www.usenix.org/conference/fast21/presentation/pan

[65] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[66] Alexander Petrov and Yifei Zhang. 2020. Ai @scale 2020: Mastercook: Large scale concurrent model development in ADS ranking. https://atscaleconference.com/videos/ai-scale-2020-mastercook-large-scale-concurrent-model-development-in-ads-ranking/

[67] Raghu Prabhakar and Sumti Jairath. 2021. SambaNova SN10 RDU:Accelerating Software 2.0 with Dataflow. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. 1–37. https://doi.org/10.1109/HCS52781.2021.9567250

[68] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 123–137. https://doi.org/10.1145/2785956.2787472

[69] Sebastian Ruder. 2017. An overview of gradient descent optimization algorithms. arXiv:1609.04747 [cs.LG]

[70] Geet Sethi, Bilge Acun, Niket Agarwal, Christos Kozyrakis, Caroline Trippel, and Carole-Jean Wu. 2022. RecShard: Statistical Feature-Based Memory Optimization for Industry-Scale Neural Recommendation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS 2022)*.

[71] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1802–1813. https://doi.org/10.1109/ICDE.2019.00196

[72] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Sigcomm '15*.

[73] Akshitha Sriraman and Abhishek Dhanotia. 2020. Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 733–750. https://doi.org/10.1145/3373376.3378450

[74] TensTorrent. 2022. Tenstorrent. https://tenstorrent.com/

[75] Tesla. 2022. Tesla Artificial Intelligence. https://www.tesla.com/AI

[76] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: A Warehousing Solution over a Map-Reduce Framework. *Proc. VLDB Endow.* 2, 2 (Aug. 2009), 1626–1629. https://doi.org/10.14778/1687553.1687609

[77] Lipeng Wang, Songgao Ye, Baichen Yang, Youyou Lu, Hequan Zhang, Shengen Yan, and Qiong Luo. 2020. DIESEL: A Dataset-Based Distributed Storage and Caching System for Large-Scale Deep Learning Training. In *49th International Conference on Parallel Processing - ICPP* (Edmonton, AB, Canada) *(ICPP '20)*. Association for Computing Machinery, New York, NY, USA, Article 20, 11 pages. https://doi.org/10.1145/3404397.3404472

[78] Yu Wang, Gu-Yeon Wei, and David Brooks. 2020. A Systematic Methodology for Analysis of Deep Learning Hardware and Software Platforms. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 30–43. https://proceedings.mlsys.org/paper/2020/file/c20ad4d76fe97759aa27a0c99bff6710-Paper.pdf

[79] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. 2021. RecSSD: Near Data Processing for Solid State Drive Based Recommendation Inference. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 717–729. https://doi.org/10.1145/3445814.3446763

[80] Carole-Jean Wu, Robin Burke, Ed Chi, Joseph A. Konstan, Julian J. McAuley, Yves Raimond, and Hao Zhang. 2020. Developing a Recommendation Benchmark for MLPerf Training and Inference. *CoRR* abs/2003.07336 (2020). https://arxiv.org/abs/2003.07336

[81] Doris Xin, Hui Miao, Aditya Parameswaran, and Neoklis Polyzotis. 2021. *Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities.* Association for Computing Machinery, New York, NY, USA, 2639–2652. https://doi.org/10.1145/3448016.3457566

[82] Chih-Chieh Yang and Guojing Cong. 2019. Accelerating Data Loading in Deep Neural Network Training. *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)* (Dec 2019). https://doi.org/10.1109/hipc.2019.00037

[83] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 1–14.

[84] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[85] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 423–438. https://doi.org/10.1145/2517349.2522737

[86] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. 2019. AIBox: CTR Prediction Model Training on a Single Node. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management* (Beijing, China) *(CIKM '19)*. Association for Computing Machinery, New York, NY, USA, 319–328. https://doi.org/10.1145/3357384.3358045

[87] Y. Zhu, F. Chowdhury, H. Fu, A. Moody, K. Mohror, K. Sato, and W. Yu. 2018. Entropy-Aware I/O Pipelining for Large-Scale Deep Learning on HPC Systems. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 145–156. https://doi.org/10.1109/MASCOTS.2018.00023

[88] Y. Zhu, W. Yu, B. Jiao, K. Mohror, A. Moody, and F. Chowdhury. 2019. Efficient User-Level Storage Disaggregation for Deep Learning. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. 1–12. https://doi.org/10.1109/CLUSTER.2019.8891023