Tectonic-Shift: A Composite Storage Fabric for Large-Scale ML Training

Mark Zhao, *Stanford University and Meta;* Satadru Pan, Niket Agarwal, Zhaoduo Wen, David Xu, Anand Natarajan, Pavan Kumar, Shiva Shankar P, Ritesh Tijoriwala, Karan Asher, Hao Wu, Aarti Basant, Daniel Ford, Delia David, Nezih Yigitbasi, Pratap Singh, and Carole-Jean Wu, *Meta;* Christos Kozyrakis, *Stanford University*

https://www.usenix.org/conference/atc23/presentation/zhao

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Open access to the Proceedings of the 2023 USENIX Annual Technical Conference is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

Tectonic-Shift: A Composite Storage Fabric for Large-Scale ML Training

Mark Zhao[1,2], Satadru Pan[2], Niket Agarwal[2], Zhaoduo Wen[2], David Xu[2], Anand Natarajan[2], Pavan Kumar[2], Shiva Shankar P[2], Ritesh Tijoriwala[2], Karan Asher[2], Hao Wu[2], Aarti Basant[2], Daniel Ford[2], Delia David[2], Nezih Yigitbasi[2], Pratap Singh[2], Carole-Jean Wu[2], Christos Kozyrakis[1]
[1]*Stanford University,* [2]*Meta*

Abstract

Tectonic-Shift is the storage fabric for Meta's production machine learning (ML) training infrastructure. Industrial storage fabrics for ML need to meet both the intensive IO and high-capacity storage demands of training jobs. Our prior storage fabric, *Tectonic*, used hard disk drives (HDDs) to store training data. However, HDDs provide poor IO-per-watt performance. This inefficiency hindered the scalability of our storage fabric, and thus limited our ability to keep pace with rapidly growing training IO demands.

This paper describes our journey to build and deploy *Tectonic-Shift*, a composite storage fabric that efficiently serves the needs of our training infrastructure. We begin with a deep workload characterization that guided an extensive hardware and software design space exploration. We then present the principled design of *Tectonic-Shift*, which maximizes storage power efficiency by combining *Shift*, a flash storage tier, with *Tectonic*. *Shift* improves efficiency by absorbing reads using IO-efficient flash, reducing required HDD capacity. *Shift* maximizes IO absorption via novel application-aware cache policies that infer future access patterns from training dataset specifications. *Shift* absorbs $1.51-3.28\times$ more IO than an LRU flash cache and reduces power demand in a petabyte-scale production *Tectonic-Shift* cluster by 29%.

1 Introduction

The success of industrial machine learning (ML) training is enabled by highly efficient and scalable infrastructures that store and feed massive amounts of training data to datacenter-scale training clusters [27, 31, 44, 46, 68]. At Meta, we deploy training clusters, each with of thousands of GPUs, across many datacenters in order to meet our ML training demands [42]. Each cluster requires a storage fabric capable of storing exabytes of data and serving reads at tens of terabytes per second.

Our prior storage fabric was *Tectonic*, Meta's exabyte-scale distributed file system [50]. Each *Tectonic* instance is backed by a cluster of disaggregated hard disk (HDD) storage nodes. To feed trainers, we needed to provision each *Tectonic* cluster with HDDs to provide *both* sufficient storage capacity for training datasets and enough IO capacity to meet the read bandwidth demands of all trainers in the datacenter. The significant and increasing IO requirements of training accelerators resulted in a large imbalance between IO and storage demands compared to what is afforded by modern HDDs. We needed to provision an order of magnitude more storage capacity to meet trainers' IO demands than to store datasets. This storage inefficiency expended a large portion of each datacenter's power budget — *modern ML storage fabrics often require more power than trainers themselves* [68] — which constrained the scalability of our training infrastructure.

This paper chronicles our journey to improve the power efficiency of our production storage fabric for IO-bound ML training workloads. We begin with a hardware design space exploration and show that traditional homogeneous storage fabrics (HDD or otherwise) cannot meet the imbalanced storage and IO demands of ML training without resource over-provisioning. An ideal storage solution should combine multiple storage media in a *composite storage fabric* to balance storage and IO capacity. It can efficiently meet IO demands by serving most IOPS from IO-efficient (high bytes/s per watt) devices, e.g., flash, while relying on storage-efficient (high bytes per watt) devices, e.g., HDDs, to meet storage demands.

However, simply deploying a composite storage fabric does not beget high efficiency. It must hold the *right data* in IO-efficient devices at the *right time* — exploiting data locality via caching. We present a software design space exploration, guided by a deep characterization of our production ML training workloads, showing that current cache systems do not capture the data reuse characteristics of these workloads. General-purpose software flash and DRAM caches are designed for web-based workloads with trillions of small requests such as content delivery networks, social graphs, key-value stores, and databases [1, 6, 7, 9, 13, 38, 43, 45, 55, 56, 66]. Meanwhile, ML training jobs issue a small number of massive scans over petabytes of data, resulting in scan and churn patterns that easily thrash an LRU cache [52]. Alternatively, current ML-

specific storage systems [16, 23, 30, 32, 41, 61, 71] are ineffective because existing solutions have been designed for small-scale deployments with highly-synchronized training jobs reading multiple epochs of the same static data. Meanwhile, large-scale production training environments consist of highly asynchronous, single-epoch training jobs reading varying subsets of continuously-updated datasets.

While cache systems leveraging composite storage have been widely studied and deployed, our hardware and software design space exploration elucidates the need for a unique combination of techniques tailored to our ML workloads. To this end, we built *Tectonic-Shift*, a composite storage fabric that improves storage efficiency by balancing storage and IO capacity across HDDs and flash. We present *Tectonic-Shift* and several guiding design principles that make it deployable and effective across our datacenters: a) *Transparent. Tectonic-Shift* presents the same APIs as the *Tectonic* File System, requiring no user knowledge or application changes. *Tectonic-Shift* combines *Shift*, a flash storage tier that aims to maximize IO absorption, with each HDD *Tectonic* cluster. b) *Simple. Shift* is built on top of CacheLib [6], and deploying *Shift* requires no changes to other storage services such as *Tectonic*'s Metadata Layer. c) *Scalable. Shift* is fully decentralized, consisting only of disaggregated flash storage nodes, each using local dynamic cache policies that adjust to observed load. d) *Intelligent.* While *Tectonic-Shift* is transparent to users, it understands application information from training job specifications, such as the list of table partitions that comprise the job's dataset. We present novel cache mechanisms that leverage this information to improve the performance of *Shift* by inferring training jobs' future data access patterns.

We demonstrate how these principles allow *Shift* to absorb $1.51-3.28\times$ IO than an LRU-based flash cache on a mix of representative training workloads, all while managing flash endurance limits. Furthermore, we present results on our petabyte-scale production tiers, serving real ML training jobs, showing how *Tectonic-Shift* can save 29% of power relative to using HDDs alone for training data. We close with a discussion of lessons learned in deploying *Tectonic-Shift* and several promising areas of future exploration. In summary, we make the following contributions.

- We provide an in-depth hardware and software design space exploration of storage systems for ML training jobs, guided by a characterization of our production workloads.
- We present the principled design of *Tectonic-Shift*, which combines *Shift*, a flash storage tier, with each *Tectonic* cluster to improve the overall efficiency of Meta's storage fabric.
- We describe novel cache policies employed by *Shift* that predict and optimize for future data access patterns derived from training job specifications.
- We show detailed production evaluation results. *Shift* absorbs $1.51-3.28\times$ more IO than LRU. *Tectonic-Shift* improves the power efficiency of our storage fabric by 29%.

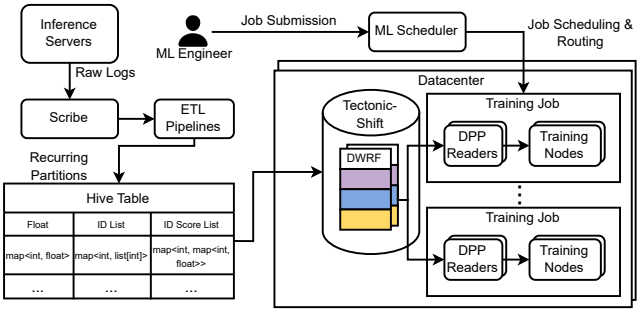


Figure 1: Overview of Meta's data storage and ingestion (DSI) pipeline. *Tectonic-Shift* is the durable storage fabric for training data in each datacenter.

2 ML Data Storage and Ingestion Background

Deep learning recommendation model (DLRM) training dominates our ML infrastructure demands [2], requiring significant data storage and ingestion (DSI) capacities to manage structured datasets [68]. Thus, we primarily focus on DLRM workloads in this paper, and we discuss extending *Tectonic-Shift* to other ML domains and non-ML workloads in Section 7. Figure 1 shows how the DSI pipeline continuously generates, stores, and ingests DLRM training data.

Data Generation. Fresh data is needed to ensure model accuracy [18]. We continuously generate training samples from inference requests served by our production fleet. When a given host serves an inference request, it logs a snapshot of the relevant *features* of the requester (e.g., a user's set of liked pages) and the outcome of the *event* corresponding to the inference request (e.g., if a user likes the recommendation). These logs are continuously published to Scribe [26], Meta's global distributed messaging system. A training data pipeline, corresponding to a set of extract-transform-load (ETL) jobs (e.g., Spark [67]), consume these logs by joining and labeling them to form structured training samples.

Dataset Storage. Each pipeline's training samples are stored in a corresponding Hive [57] table. Tables are constantly updated with new time-based partitions of fresh data, generated by each pipeline with a regular cadence (e.g., hourly). Old partitions regularly expire and are deleted. Each table is replicated to all datacenters with training clusters, and each partition is stored as columnar DWRF [21] files (similar in format to ORC [14]) in a new directory in each respective datacenter's *Tectonic* File System. Training jobs read from their local *Tectonic* instance.

We adopt a common schema across our training tables to ensure interoperability across models [68]. Specifically, all features are stored in a small number of map columns and comprise the majority of each row ($>99\%$ of bytes). Each column maps multiple integer feature IDs to the row's corresponding value for that feature (e.g., a float for a dense feature column or lists/maps for a categorical feature column).

Data Ingestion. Training jobs are submitted to a global queue

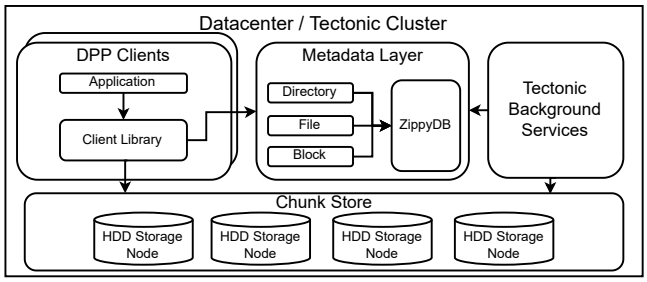


Figure 2: Overview of the *Tectonic* File System. Directory, file, and block metadata operations are served by a Metadata Layer. Clients directly read chunks from HDD-based Chunk Store nodes.

by ML engineers and are scheduled and routed to a specific datacenter when capacity allows. When each job is scheduled, it allocates a set of training nodes (Trainers) [42] and a set of Data PreProcessing (DPP) Readers [68] from the datacenter's training cluster. Trainers are equipped with GPUs and perform the actual training, continuously ingesting tensors from Readers. Readers are general-purpose CPU nodes that continuously read raw bytes from the *Tectonic* File System, reconstruct minibatches of samples from the bytes, and preprocess each minibatch into tensors.

Specifically, Readers read data based on a training job's *dataset*, specified by an ML engineer. The dataset contains a *list of table partitions* and a *list of feature IDs*. Throughout the lifetime of the training job, Readers will continuously ingest (disjoint) minibatches of samples, filtering out unused features in each sample, until the Readers have collectively read all samples from the specified partitions. To read the appropriate bytes from *Tectonic*, Readers map each partition to a set of *Tectonic* files by querying the Hive Metastore [57,58]. Readers then scan each file and progressively read samples by issuing *Tectonic* reads. Readers push filtering to storage, referencing metadata within footers of the file to selectively read bytes corresponding to features specified by the dataset [68].

***Tectonic* File System.** Figure 2 shows the architecture of the *Tectonic* cluster (sans *Shift*) backing each *Tectonic* File System instance. Files are divided into *blocks* (typically 72 MiB) representing a logical array of bytes. *Tectonic* further divides blocks into smaller *chunks* (typically 8 MiB) and durably encodes each via replication or Reed-Solomon (RS) encoding [51]. Chunks are distributed across the cluster's Chunk Store, backed by a number of HDD storage nodes.

Readers directly read data from storage nodes using the *Tectonic* Client Library. The Client Library exposes a file `pread` interface to clients. For each read, the Client Library issues requests to specific chunks on storage nodes and performs reconstructions if necessary. The Client Library obtains chunk mappings and any directory and file metadata (e.g., directory `ls`) via queries to a hash-sharded Metadata Layer built on ZippyDB [37]. DPP Readers optimize for HDD seeks and coalesce reads into large $O(1MB)$-sized IOs [68].

Table 1: Storage power requirements for an HDD, flash, and ideal composite cluster, assuming 100 PB and 10 TB/s storage and IO demand. We show required power to meet storage-only, bandwidth-only, and both requirements, normalized to HDD storage-only.

	Storage Req.	IO Req.	Storage & IO Req.
HDD Cluster	1.00	9.92	9.92
Flash Cluster	6.53	1.88	6.53
HDD + Flash	1.00	1.88	2.69

3 Production ML Storage Design Space

This section explores why we could not efficiently scale *Tectonic* to meet the IO bandwidth that our training clusters increasingly demand. We present various hardware and software design space explorations that led us to *Tectonic-Shift*, guided by a characterization of our production ML training jobs.

3.1 Hardware Design Space

We first evaluated different storage hardware options, summarized in Table 1. Specifically, we used our HDD [3] and flash [8] server specifications to calculate the power (watts) required by the number of HDD, flash, or HDD + flash servers (rows) to supply 100 PB of storage, 10 TB/s of read bandwidth, or both (columns). These demands are representative of our workloads [42,68], and we must provide both sufficient storage *and* IO capacity. The HDD + flash analysis used only HDDs to supply 100 PB of storage and only flash to supply 10 TB/s of IO. In the storage and IO case, we used HDDs to supply storage capacity and flash to supply IO capacity, discounting the IO capacity supplied by the HDDs. We focused on power because it is the primary budget and optimization metric for services across our fleet [68]. We normalized results to the HDD, storage-only case.

Option 1: HDD-Only (Status Quo). Our first option to meet IO demand was to continue provisioning more HDD storage nodes into each *Tectonic* cluster. This would linearly scale IO capacity as chunks are distributed across HDDs evenly. Unfortunately, this option would require us to provision 9.92× more storage capacity than necessary — $1.6EB$ of disks assuming $RS(9,6)$! Furthermore, since our IO demand is growing 2× as fast as storage demand [68], this option is unsustainable.

Option 2: Flash-Only. We also considered using a flash storage tier [28] for our training datasets. Flash trades off storage-efficiency for IO-efficiency. Compared to HDDs, A flash cluster would need 5.28× less power to meet IO demand, but 6.53× more power to meet storage demand. Meeting both demands is more efficient using flash, but there is a significant over-provisioning of IO capacity, making this sub-optimal.

Option 3: Composite Storage. Relying on a single storage hardware inherently precludes us from balancing storage and IO capacity. An ideal cluster would use *both* a storage-efficient device and IO-efficient device together, provisioning enough of each to meet their respective demands. HDDs are

Figure 3: IO bandwidth demand across 85 tables over the course of one day. Training tables exhibit a power law in popularity.

an ideal storage-efficient device due to their density. We considered DRAM and flash for our IO-efficient device.

Option 3.1: HDD + DRAM. We decided against using only DRAM, as a DRAM storage node would be bound by the NIC throughput (e.g., 100 Gbps) as opposed to DRAM throughput. Modern SSDs can provide $O(1GB/s)$ of read bandwidth at $O(1W)$ of power [11], allowing a flash storage node to provide the same IO capacity in roughly the same power footprint as a DRAM storage node. Meanwhile, flash storage nodes have significantly higher storage capacities ($O(10TB)$), greatly improving cache performance.

Option 3.2: HDD + Flash. We opted to use flash as our underlying IO-efficient device. Table 1 shows how an ideal composite cluster would allow us to provision only 1.69× flash-based power to meet IO demand plus 1.00× HDD-based power to meet storage demand, reducing the power footprint of our storage tier by 3.69× compared to Option 1.

However, Option 3.2 assumes that flash servers are able to meet the bulk of IO demand by holding a popular subset of bytes. Our next challenge was designing a system that could intelligently manage the contents of each flash server to maximize its IO absorption. Our first option was to create both a flash and HDD Chunk Store within the same durable storage fabric. The *Tectonic* Metadata Layer would move blocks between flash and HDD based on read demand. This option has several drawbacks. It a) requires extra RS encoding overheads on flash, b) adds metadata overheads due to block location updates, and c) requires significant changes to the Metadata Layer to support sub-block granularities (due to byte-range popularities, to be discussed in Section 3.2.1). For these reasons, we decided to use flash as the foundation for a metadata-less, non-durable cache. We present the design space exploration of this software cache next.

3.2 Software Design Space

3.2.1 Production ML Workload Characterization

We begin by characterizing our production ML training jobs, which present a uniquely challenging cache workload.

Row-wise Reuse. Training samples (rows) exhibit a skewed popularity across training jobs. Figure 3 shows the IO demand targeting 85 tables over the course of one day. We run many ML model types in production, and ML engineers continuously train and experiment on each model type with varying

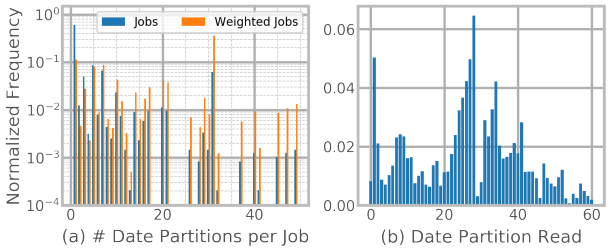


Figure 4: (a) Normalized histogram showing # of date partitions read by each training job, with orange bars weighing jobs by the number of partitions read. (b) IO demand across date partitions over a one-day period, with day 0 being the most recent partition.

popularities. Since each model type typically uses a distinct table, this variation manifests in tables' IO demands. There is a distinct power-law in table popularity with a long tail.

Furthermore, training jobs typically read a subset of table partitions, as models can typically reach convergence before all rows are exhausted. For a similar reason, each training job only reads its specified rows once (i.e., one epoch). Figure 4(a) shows a distribution for a popular table[1] of the number of date partitions[2] read by training jobs in one day. Most jobs read only a few partitions. However, when we consider IO demand by weighing the impact of each job by the number of date partitions it reads (the orange bars), we see that the majority of IO demand comes from jobs that read 20 or more partitions.

It is also important to understand *which* partitions training jobs typically read. Figure 4 shows a normalized histogram of IO demand over the popular table's date partitions over the course of one day. Partitions do not exhibit flat popularity, but instead show multiple modalities. A large fraction of traffic reads the most recent date partition. This is typical of "recurring" jobs that keep the model up-to-date and exploratory jobs that use the freshest data. There also exist multiple groups of popular date partitions, where multiple larger-scale training jobs use similar date ranges to ensure comparable results.

The above graphs show multiple important characteristics. a) Row reuse is solely across single-epoch training jobs. b) The active working set size of all training jobs is massive. There are over 85 active tables, each containing $O(1-10PB)$ of samples [68]. c) Most IO demand comes from jobs that read tens of date partitions. These jobs have PB-scale working sets due to $O(100TB)$-sized date partitions [68]. There are also many smaller jobs that read a few date partitions. d) Date partitions exhibit varying popularity, with popularity changing over time as date partitions are generated and deleted.

Column-wise Reuse. Columns (i.e., features) also exhibit a distribution in popularity, as training jobs typically use a subset of all features due to hardware (e.g., GPU memory) constraints. Figure 5(a) shows an analysis of 1265 production training jobs that read a specific date partition of the popular

[1]Where relevant, we characterize this same table throughout this section.
[2]A date partition consists of all training samples generated in a given day.

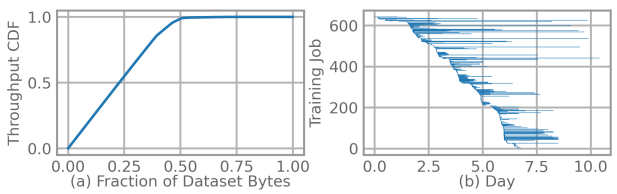


Figure 5: (a) CDF showing distribution of stored feature bytes to IO served to 1265 training jobs reading a single date partition. (b) Time-series of training jobs reading a popular table over one week.

table. The x axis shows a distribution of the stored bytes, ordered by read popularity. The y axis shows the fraction of total IO bandwidth served by most popular x fraction of bytes. Over 75% of bytes are read at least once. However, roughly half of all bytes, corresponding to popular features, serve almost all IO demand.

This has important implications for a cache system, as features are stored into columnar byte streams within large DWRF files. Specifically, row-wise popularity corresponds to *file-popularity*, and column-wise popularity corresponds to *byte-range popularity* within each file. A cache system must effectively capture both dimensions.

Temporal Behavior. Cache systems must also account for the unique temporal characteristics of training jobs. Figure 5(b) shows a time-series plot of 642 jobs, launched over one week, that read the popular table. Each horizontal bar reflects the lifetime of each job, during which it reads the samples and features specified in its dataset. We observe that first, training jobs are largely asynchronous. Cache systems cannot solely optimize for highly synchronized jobs (e.g., hyperparameter tuning) and assume high temporal locality. Secondly, data reuse is expressed across a small number of training jobs, unlike the billions of requests common to web-based workloads [6]. Finally, each training job can run from hours to days, requiring a large storage and temporal footprint.

3.2.2 Cache Software Design Space Exploration

With our workload characteristics in mind, we evaluated various system architectures to manage our flash storage tier.

Option 1: General-purpose Software Caches. We built CacheLib at Meta as a general-purpose cache engine to support caches for datacenter applications such as key-value stores, databases, CDNs, and social graphs [6]. CacheLib offers LRU and FIFO eviction policies over flash, and random and reject first admission policies to manage flash endurance. Our first option was to simply deploy a cluster of flash storage nodes, each managed by CacheLib, to cache file byte ranges.

Unfortunately, ML training workloads exhibit patterns that general-purpose cache policies fail to handle. Our characterization showed a long tail of jobs that read unpopular tables and partitions. However, each job can still have working sets up to tens of petabytes, potentially larger than the entire cache itself. These massive and long-running *scans* can easily evict the entire cache, reducing hits on popular items. Furthermore, even popular training samples are susceptible to *churn*, where each sample is repeatedly evicted and inserted into cache. Churn occurs because a) there are relatively few training jobs in each datacenter, b) data reuse occurs with a relatively long duration between jobs (Figure 5), and c) working set sizes exceed our cache capacities. Scans and churns are well-known antagonist cache patterns [52], motivating the need for specialized and domain-specific admission and eviction policies.

Option 2: ML-specific Caches. We also considered techniques for building an ML-specific cache, inspired by recent work [16, 23, 30, 32, 41, 61, 71], that allows applications to explicitly cache samples in high-bandwidth storage. While such caches can optimize policies for ML workloads, they face several disadvantages. First, current ML caches employ techniques that require assumptions not representative of our workloads, limiting their effectiveness. For example, they cache entire files (e.g., images), and they rely on a large amount of intra-job data reuse across multiple epochs and inter-job data reuse across highly concurrent hyperparameter tuning jobs. Meanwhile, feature popularity requires our cache to store byte ranges within files, and our workloads only exhibit inter-job data reuse with highly asynchronous workloads. Secondly, an application-controlled cache introduces security concerns due to the need to handle access to and deletion of multiple copies of data. Finally, ML caches require end-user efforts to adopt, hindering both our deployment velocity, and more importantly, the productivity of ML engineers.

Option 3: A Transparent, Application-aware Cache. Our final cache design combined benefits from both software and ML caches. We focused on policies that provide the transparency and generalizability of software caches and the application-level awareness of ML caches. Our characterization highlighted key opportunities. Specifically, not only do training jobs tend to favor specific rows and columns, their dataset specifies *which* features, tables, and partitions the job will deterministically read throughout its lifetime.

We next explore how our entire design space exploration yielded a principled design of *Tectonic-Shift*. Section 5 then describes how we leverage policies that infer future access patterns from dataset specifications to minimize cache contention and to maximize the read IO absorbed by the cache.

4 *Tectonic-Shift* Architecture

4.1 *Tectonic-Shift* Design Principles

Figure 6 shows the architecture of *Tectonic-Shift*. We designed *Tectonic-Shift* around four key design principles.

Transparency: Tectonic-Shift combines *Shift*, a flash storage tier, in front of each HDD *Tectonic* cluster transparently. Each *Tectonic-Shift* cluster serves read requests for all training workloads in its respective datacenter. It exposes the same

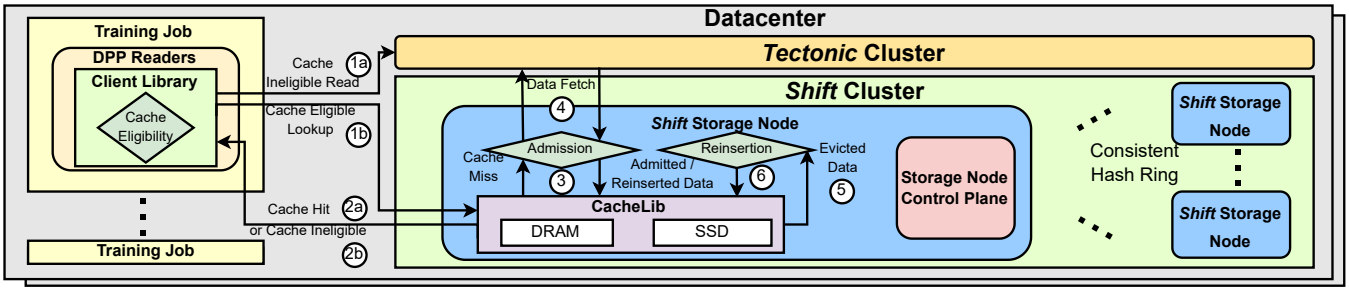


Figure 6: The block diagram of *Tectonic-Shift*, with numbered arrows depicting the path of each read request.

APIs and semantics as our current *Tectonic* File System. Transparency was important for two reasons. First, we avoided exposing storage decisions to ML engineers, as doing so may lead to inefficient configurations and hinder their productivity. It also eased deployment, allowing us to progressively roll out *Shift* and re-balance *Tectonic* clusters under the hood. Secondly, because most of Meta's storage services rely on *Tectonic*, a general API would allow us extend *Shift* to new customers that could become IO-bound (see Section 7).

Simplicity: We kept *Shift* simple and robust by reusing as much infrastructure as possible. *Shift* can only be accessed via the Client Library, which uses *Tectonic*'s Metadata Layer. This simplifies security since all access controls are validated before reaching *Shift*. *Shift* also uses the fact that training data is stored in immutable, *sealed* blocks [50] to avoid managing cache invalidations or mutations. Each *Shift* node uses CacheLib as its internal caching engine, allowing us to harness resources from the myriad teams that rely on CacheLib.

Scalability: We built *Shift* to be effective at any deployment size, quickly deployable, and easily scalable to meet the demands of each datacenter. *Shift* is fully decentralized, consisting of only flash storage nodes placed in a consistent hash ring. Cache decisions are only made locally to each storage node and dynamically adjust based on observed load.

Intelligence: Finally, based on Section 3, *Shift* must adapt to the unique workload characteristics of ML training jobs. Section 5 explores how we built intelligent cache policies on top of CacheLib. These policies infer each training job's data access pattern based on its initial dataset specification, allowing each *Shift* node to maximize IO absorption based on both historic and expected future data access patterns.

4.2 The Life of a *Tectonic-Shift* Read

Client Library. As discussed in Section 2, for each training job, DPP Readers collectively scan through the specified partitions, with each Reader individually reading separate splits of rows. Each partition is mapped to a distinct file system directory, and Readers directly read data corresponding to used features from files in the respective directories. Readers obtain file handles by querying the *Tectonic* Metadata Layer (Figure 2). Mappings from features and rows to file byte ranges are decoded by Readers from file footers. Each Reader issues reads via a `pread` Client Library API call, which returns `count` bytes starting at `offset` within a file.

Figure 6 shows how the Client Library handles each `pread`. It first decomposes the `pread` into a set of block reads by querying the *Tectonic* File Layer. The Client Library then checks if each block read is cache eligible. Cache *in*eligible reads directly read each block range from the *Tectonic* Chunk Store (1a). Cache eligible reads directly issue a `get(blockId, offset, length)` for each block to the *Shift* cluster (1b). We piggyback a number of *tags* with each `get` request that associate each request with relevant metadata, such as the file path and training job ID, to be used by *Shift* policies. The *Shift* cluster consists of a number of flash *Shift* Storage Nodes (SNs) placed in a consistent hash ring [25]. The Client Library maps each `get` request to a *Shift* SN based on `hash(blockId)`. `get` requests either return data for the corresponding block (2a), or return a cache miss (2b). The Client Library reads missed blocks from the *Tectonic* cluster (1a). Once all blocks are fetched, the Client Library returns the results of the `pread` to the caller.

***Shift* Storage Node Data Plane.** *Shift* uses CacheLib [6] within each SN to manage both DRAM and flash. We break up each *Tectonic* block into fixed-size *segments*, which are the objects that we place into CacheLib. Blocks are typically 72 MiB; we discuss segment sizes in Section 5.2.

The SN breaks up each `get` request's range into segments. If all segments are present in cache, the SN simply returns the requested data (2a). Otherwise, *Shift* implements two critical policies on top of CacheLib. First, if any segments are missing, the SN decides if the segment is *admitted* (i.e., allowed) into cache (3). If any segment is not admitted, the SN returns a cache ineligible miss to the Client (2b). Otherwise, the SN will fetch admitted segments from the *Tectonic* cluster and insert them into cache (4). The SN then returns data corresponding to the `get` request (2a). Secondly, any cache insertions will potentially result in segments being evicted from the cache (5). For each evicted segment, the SN can optionally *reinsert* the segment into cache (6), potentially avoiding a cache miss if the evicted segment is accessed in the near future.

***Shift* Abstractions and Guarantees.** The primary goal of

Shift is to serve IO bandwidth corresponding to popular bytes, reducing IO to *Tectonic*'s Chunk Store. We rely on and do not change the semantics of the *Tectonic* File System.

Specifically, the *Tectonic* File System provides append-only semantics. Data pipelines will *seal* blocks; we do not have to handle modifications in *Shift*. *Shift* SNs act only as a part of the data plane. To keep file system operations centralized and scalable, we rely on the *Tectonic* cluster's Metadata Layer for all metadata operations. Thus, *Shift* does not expose a `put` API. Inserts into cache are only made for missed `get` requests. Accesses to blocks are consistent since *Shift* SNs can only be accessed by the *Tectonic* Client Library, which first references the Metadata Layer for file-to-block mappings, preventing reads to renamed or deleted files. Similarly, unauthorized reads are prevented as any the Client Library performs ACL checks for each block before issuing reads to *Shift*.

Our proposed design also allows *Shift* to be inherently fault tolerant. *Shift* contains no centralized state. *Shift* relies on the fault tolerant *Tectonic* Metadata Layer [50] for metadata operations. Cache policies are made local to each SN, allowing other SNs to proceed unhindered in the event of a SN failure. Furthermore, *Shift* serves only reads, and Clients will default to *Tectonic* reads (e.g., after a timeout) if a given SN fails.

5 Application-Aware Cache Policies

Building on our design principles, our key insight is to instrument *Shift* with intelligent policies that maximize the IO absorbed from *Tectonic*. These policies transparently adapt to workloads by leveraging both historic and application information, ensuring that only segments with high reuse across training jobs are kept in each SN. Specifically, each *Shift* SN contains a Control Plane that implements an **admission** and **reinsertion** policy. We incorporate a **cache eligibility** policy in the Client Library to reduce RPC pressure and avoid requests to *Shift* from "uncacheable" workloads.

We focused on building a flexible SN Control Plane that aggregates the necessary metadata (including application information) to define and inform highly configurable policies, allowing us to constantly tune and improve performance. We prioritized admission and reinsertion policies as opposed to eviction policies such as LRU because a) we can prevent significant thrashing due to the scan and churn (Section 3) patterns common in our workloads, and b) we can control write rates to flash in order to manage flash endurance constraints. Meanwhile, our policies are built on top of CacheLib, and we use CacheLib's provided eviction policies (LRU or FIFO) after admission into cache.

As discussed in Section 4, each *Shift* SN acts as an independent entity, serving only requests on its portion of a consistent hash ring. Thus, the overall goal of *each Shift* SN is to maximize the absorbed IO locally — doing so maximizes the aggregate IO absorbed by the entire *Shift* cluster. We define the absorbed IO at each SN as the bandwidth of successful

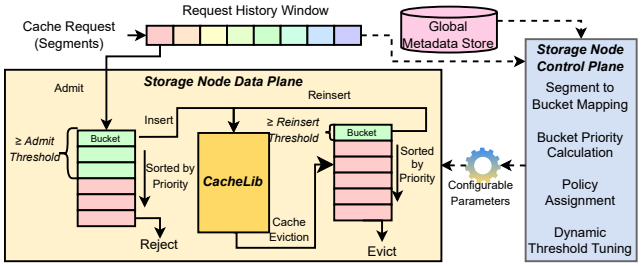


Figure 7: Overview of *Shift* SN Control and Data Plane. The Control Plane dynamically tunes Data Plane cache policies.

Table 2: Table listing configurable parameters in *Shift*.

Configurable Parameter	Meaning
$MapSegment(s)$	Mapping function from segment to bucket.
$BucketPriority(b)$	Function to calculate priority of bucket b.
$AdmitThreshold$	Dynamic scalar threshold to admit buckets.
$ReinsertThreshold$	Dynamic scalar threshold to reinsert buckets.
$BucketRefreshTime$	Update period for bucket policies.
$RHWSize$	Size of request history window logs to keep.

`get` requests returned to clients, minus the bandwidth of data fetches to the *Tectonic* cluster due to data misses. Importantly, non-admitted cache requests do not impact absorbed IO, and reinserted segments do not contribute to fetches to *Tectonic*.

5.1 Admission and Reinsertion

Figure 7 shows an overview of the Control and Data Plane running in each SN. The Control Plane dynamically directs the Data Plane to admit or reinsert segments into the cache upon a cache miss or eviction, respectively, using *historical* and *application* metadata. These decisions are made based on the configurable parameters summarized in Table 2.

Mapping a Segment to a Bucket. The Control Plane statically maps each segment s to a *bucket* b using the tags attached to each read request, based on a configurable function $MapSegment(s)$. Intuitively, each bucket represents a collection of segments that will likely be accessed together and connects to a logical grouping within the application.

For example, we typically use a segment's corresponding directory as its default bucket mapping, as each training job specifies a set of partitions to read and will scan through files within each partition's directory. Furthermore, we can correlate data reuse across multiple jobs based on their dataset partitions (and thus directories). While finer-grained bucket mappings such as files or features (i.e., file byte ranges) are possible, directories provide sufficient granularity since jobs mostly read similar features within each file (Section 3).

Assigning a Policy to Each Bucket. Each bucket is assigned a binary admission and reinsertion policy. The Data Plane simply admits/rejects (on miss) or reinserts/evicts (on eviction) each segment depending on its bucket's current policy. The Control Plane will admit or reinsert a bucket if the bucket's current *BucketPriority(b)* is greater than *AdmitThreshold* or

ReinsertThreshold, respectively. Bucket policy assignments are updated every *BucketRefreshTime*, a configurable parameter. Buckets should be updated frequently enough to react to changes in reading patterns; we use a default of 10 seconds.

Deriving a Bucket's Priority. *BucketPriority(b)* directly determines b's admission/reinsertion policy. Intuitively, we interpret b's priority as *the number of times we expect each segment in the bucket to be read in the near future*. Higher priority buckets will be placed into the cache (via admission/reinsertion) over lower-priority buckets and thus absorb more IO in total. Our key insight is to calculate buckets' priorities using both *historical* and *future* information derived from the Request History Window (RHW) and Global Metadata Store (GMS), respectively.

Historic Priority. The RHW tracks recently observed requests (regardless of admission) over a past time period, $RHWSize$. The RHW reports the number of *unique* segments and *total* segments requested for each bucket. A *historic priority* for bucket b can be calculated as $BucketPriority(b) = TotalBytes(b)/UniqueBytes(b)$, providing the traditional cache signal which assumes that past access patterns are indicative of the future. $RHWSize$ is a configurable parameter. We tune it to capture sufficient historical data without exceeding memory capacity limits; we use a default of 6 hours.

Future Priority. The RHW also records the set of active training jobs and the set of buckets read by each training job using the job ID tag piggybacked with each request. The GMS is a set of databases which contains real-time information about the dataset specification of each training job. By combining the RHW and GMS, *we can directly derive future accesses for each training job* and thus bucket priorities. The Control Plane queries the RHW for all active training jobs and pulls each job's dataset specification from the GMS. For example, for directory-based buckets, the Control Plane derives a bucket's *future priority* as equal to the number of jobs that include the corresponding partition in its dataset, discounting any jobs that have finished reading the bucket.

In summary, the RHW captures historic access patterns and active training jobs, while the GMS associates each training job with application information about its dataset. While we presented potential historic and future policies, variations can easily be created using the RHW and GMS. For example, a potential future policy can further prioritize directories earlier in read order (and thus read sooner). Section 6 explores a *hybrid* policy combining historic and future priorities.

Threshold Tuning. The Control Plane continuously tunes the $AdmitThreshold$ and $ReinsertThreshold$ based on two factors. First, a *minimum threshold* avoids admitting unpopular workloads that can evict the entire cache. We use a minimum value that is strictly greater than 1, and we constantly tune it based on observed performance. Secondly, we implement a PID-controlled *feedback loop* to ensure that the cache admit plus reinsert rates (reported by the RHW) is strictly less than our flash endurance limits (defined by a maximum average write rate). This allows the threshold to increase beyond the minimum in response to high flash write rates, thereby admitting/reinserting fewer segments. We only tune the $AdmitThreshold$ and tie the $ReinsertThreshold$ to be a fixed offset (e.g., $AdmitThreshold + 1$) to prioritize admits over reinserts to limit write amplification due to reinserts.

An additional benefit in building an admission policy *above* CacheLib is to rate limit prior to *Tectonic* reads. While CacheLib provides a rate limiter, which selectively admits segments to flash upon DRAM eviction, it inherently requires first inserting data into DRAM. This results in unnecessary *Tectonic* reads if the data is soon to be evicted due to write endurance limits. *Shift*'s admission policy acts prior to *Tectonic* fetches, avoiding unnecessary HDD reads for rate limited data.

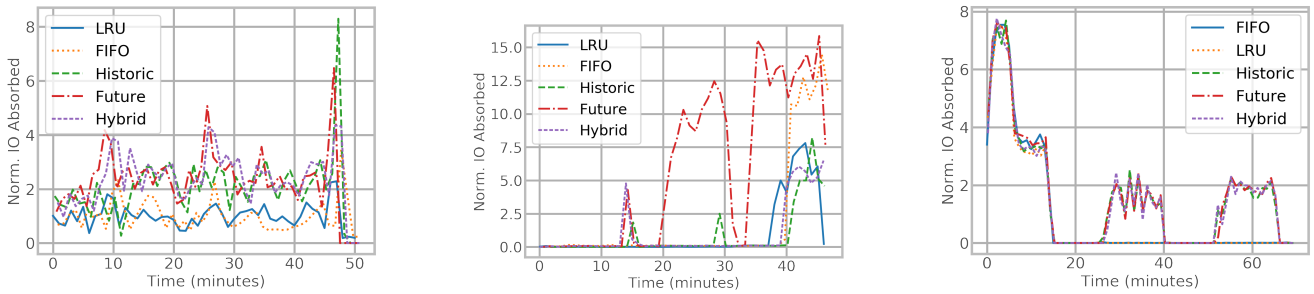
5.2 CacheLib Tuning

CacheLib offers a suite of configurable parameters that we continuously tune via a host of stress, release validation, and production tests. While prior flash caches focused on addressing write amplification caused by small objects (e.g., <1KB messages) [6, 13, 38, 56], a key difference and advantage in *Shift* is the ability to configure segment sizes. Too large segments, relative to request sizes, result in overheads since we fetch and store data in segment-granularity. On the other hand, too small segment sizes constrain both DRAM and flash due to metadata and write amplification overheads. We found that 256 KB was a good balance for our workloads. We also rely on CacheLib to optimize underlying data layouts on flash [6] to further improve flash endurance. Finally, we evaluated the available eviction policies for DRAM and flash and found that LRU works well (when combined with *Shift*'s policies); we provide further exploration in Section 6.

5.3 Client Cache Eligibility

We also incorporate a *Shift* eligibility policy at the *Tectonic* Client Library. *Tectonic* serves a diverse set of training and non-ML workloads across Meta. The primary purpose of the cache eligibility policy is to prevent customers that are not onboarded to *Shift*, as well as low-priority and uncommon (and less-cacheable) tables, from issuing lookups to *Shift*. While these read requests would likely be rejected by the SN's cache admission policy, applying a first filter significantly reduces the RPC load and memory pressure at each SN.

We currently filter out all non-ML traffic to bolster ML training capacity. Furthermore, Figure 3 shows that tables are disproportionately popular. Filtering out rarely-used tables adds another layer of protection against cache contention, since all IO can be sufficiently served from *Tectonic* HDDs. As a baseline heuristic, we filter out tables whose IO demand can be sufficiently served by the HDD capacity needed to store it. We continuously tune our filters based on demand.



(a) Synchronized workload (b) Pipelined workload (c) Sequential workload

Figure 8: IO absorbed by different policies across benchmarks, normalized to the average IO absorbed by *LRU* eviction.

Table 3: Benchmark DLRM training job workloads used for evaluation. Each job j is denoted as $\{\}_j$ and reads each partition P in specified order. Each partition P has a $\approx$ 5 TB working set.

Workload	Jobs & Partitions Read	Description
Synchronized	$\{P_1,P_2,P_3\}_1$, $\{P_4,P_5,P_6\}_2$, $\{P_1,P_2,P_3\}_3$, $\{P_7,P_8,P_9\}_4$, $\{P_1,P_2,P_3\}_5$	Multi-tenant HP tuning or exploratory jobs. Jobs are launched synchronously.
Pipelined	$\{P_1,P_2,P_3\}_1$, $\{P_2,P_3\}_2$, $\{P_3\}_3$	Long-running, pipelined jobs. Jobs are launched synchronously.
Sequential	$\{P_1\}_1$, $\{P_1\}_2$, $\{P_1\}_3$, $\{P_1\}_4$, $\{P_2\}_5$, $\{P_3\}_6$, $\{P_1\}_7$, $\{P_4\}_8$, $\{P_5\}_9$	Queued jobs that launch when training capacity is available. Jobs 1-3, 4-6, and 7-9 launch together.

6 *Tectonic-Shift* Deployment and Evaluation

Tectonic is Meta's durable storage system. It has been in production since 2015 and stores exabytes of data. *Shift* has been in production since early 2022 and is deployed at petabyte-scale alongside multiple *Tectonic* clusters serving DLRM training workloads. In this section, we evaluate *Shift*'s various caching policies compared to state of the art using a series of representative workloads, and we present results of *Shift* in production. We focus on and report absorbed IO because it is *Shift*'s top-line metric and optimization goal. A comparison of hit rates would yield analogous results since requests rates are equally distributed across SNs due to consistent hashing.

6.1 *Shift* Policy Evaluation

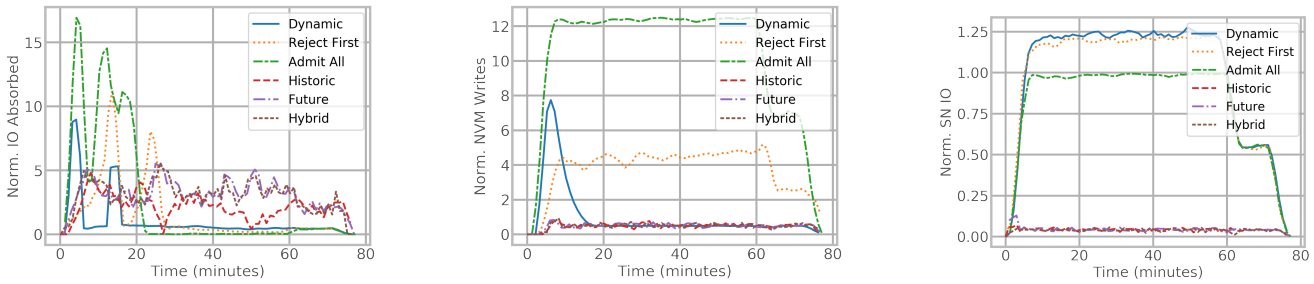
To better understand how *Shift* policies perform, we used a set of representative workload patterns shown in Table 3. Each pattern was derived from production DLRM training job traces and downsized to scale to our evaluation cluster. The *Synchronized* pattern represents a case where multiple training jobs reading the same partitions are launched at the same time (e.g., for hyperparameter tuning jobs), with training jobs reading other partitions interleaved due to the multi-tenancy of our training clusters. The *Pipelined* pattern frequently occurs in long-running jobs when users kill an under-performing job and replace it with a new model using the same dataset. The *Sequential* pattern occurs due to limited training resources, where jobs will run in separate batches as jobs finish and resources become available. For each workload, we used a set of Readers that each read from *Tectonic-Shift* at $\approx$ 1.5 GB/s.

We evaluated on a 6-node *Shift* cluster deployed with our production configuration. In each experiment, we configured nodes with different policies and evaluated each policy concurrently to ensure equal read locality; consistent hashing evenly spread requests across nodes. We used 16 GB of DRAM cache for each node. The Synchronized, Pipelined, and Sequential patterns used (1.28 TB, 5 TB, and 5 TB), and (4, 5, and 5) of total flash cache and Readers per job, respectively. Unless otherwise stated, we used directory bucket mappings and disabled reinsertion and write rate limits.

Do admission policies improve IO absorption? First, we evaluated if various *Shift* admission policies improved IO absorption across all workloads. We used the *Historic* and *Future* admission policies presented in Section 5.1, which use historic and future metadata, respectively, from the RHW and GMS to calculate a priority equal to the number of expected reads per segment. We also used a *Hybrid* admission that uses $BucketPriority_{Hybrid}(b) = max(BucketPriority_{Historic}(b), BucketPriority_{Future}(b))$. We set a minimum admit threshold of 1.1 for each policy, implementing a "reject first" policy. We used LRU eviction for each admission policy, and we compared to two baselines that only used CacheLib's *FIFO* and *LRU* eviction policies.

Figure 8 shows the IO (bandwidth) absorbed by each policy, normalized to the average IO absorbed by *LRU*. A higher IO absorption directly translates to higher *Shift* efficiency. In the Synchronized workload, *FIFO* absorbed an equal amount of IO (1.01×) as *LRU*. The *Historic* policy absorbed 2.01× more IO than *LRU*, since it was able to avoid cache thrashing induced by P_{4-9} (see Table 3) by not admitting them. The *Future* and *Hybrid* policies absorbed 2.27× and 2.32× more IO, out-performing *Historic* admission since they were also able to immediately cache $P_{1,2,3}$ without rejecting initial reads waiting for the Request History Window to populate.

For the Pipelined workload, *FIFO* was able to absorb 1.86× more IO than *LRU*, since churn caused *LRU* to evict more objects in P_3 before job 1 read P_3. *Historic* admission performed worse than *LRU*, absorbing only 0.80× IO, as it did not admit any bytes from P_3 until job 2. Since the *Future* admission policy immediately knew of P_2 and P_3's popularity,



(a) IO Absorbed (b) NVM Write Rate (c) HDD Read / Cache Fill Rate

Figure 9: Policy performance using the Synchronized workload, normalized to *Dynamic*. *Dynamic* and *Shift* write limits are set to 100 MB/s.

it was able to maximize and absorb 5.84× more IO than *LRU* by admitting them to both to cache on the first read by jobs 2 and 3. *Hybrid* admission equalled *LRU* (0.99×), since it kept admitting P_2 during job 1, expecting future reads due to historic popularity and thrashing P_3's data in cache.

Finally, we observe that for the Sequential workload, *FIFO* performed on-par (1.06× IO absorbed) with *LRU*, while *Historic*, *Future*, and *Hybrid* outperformed *LRU* equally (1.71×, 1.74× and 1.69× respectively). Specifically, for the first set of jobs (1-3), all policies saw high hit rates since only P_1 was actively read. However, for the second (4-6) and third (7-9) sets of jobs, the *Shift* policies rejected reads from P_{2-5}, avoiding contention and improving IO absorbed by P_1.

On average across all workloads, *FIFO*, *Historic*, *Future*, and *Hybrid* respectively absorbed 1.31×, 1.51×, 3.28×, and 1.67× more IO compared to *LRU*.

Can admission policies manage flash endurance? We need to limit the write rate to SSDs in order to preserve their lifetime. To study how well *Shift* policies perform under constrained write limits, we repeated the Synchronized workload while limiting each *Shift* node with an flash write limit. We compared to two state-of-the-art admission policies provided by CacheLib: a *Dynamic* flash admission policy randomly rejects writes to flash in order to maintain the specified write limit, and a *Reject First* flash admission policy rejects objects' first write to flash. We also evaluated no admission policy (*Admit All*). We used LRU eviction for all admission policies, and we configured *Shift* policies and the *Dynamic* policy to write only 100 MB/s per node. To fully evaluate the *Shift* threshold tuner, we did not set a minimum admit threshold.

Figure 9 shows the IO absorbed, flash write rate, and *Tectonic* HDD read rate for each admission policy, with each metric normalized to the average for the *Dynamic* admission policy. All admission policies absorb more IO than *Dynamic*. *Reject First* and *Admit All* absorb 1.51× and 2.66× more IO, respectively; the *Historic*, *Future*, and *Hybrid* policies absorb 2.14×, 3.07×, and 2.99× more IO, respectively.

While *Shift*'s *Future* and *Hybrid* policies performed similarly to *Admit All*, Figure 9b shows how *Admit All* required significantly more (10.38×) flash writes than *Dynamic*, exceeding our write limit. *Reject First* was similarly ineffective, requiring 3.78× more flash writes. Meanwhile, all of *Shift*'s policies matched CacheLib's write rate (within 5%, even discounting CacheLib's increased writes initially due to its counter warm-up) while outperforming its IO absorption.

Table 4: Hit rate and HDD IO (cache fills) using *Hybrid* admission with dynamically-tuned reinsertion, normalized to *Hybrid* admission without reinsertion. Write rate is limited to 1 GB/s.

	Normalized Hit Rate	Normalized HDD IO
Hybrid with Reinsertion	1.03	0.82

Furthermore, *Shift* has the advantage of avoiding excess reads from *Tectonic* HDD nodes for rejected objects, compared to CacheLib's *Dynamic* policy which always admits objects to DRAM first (and thus incurring an HDD read) before rejecting it from flash. Figure 9c shows this benefit; each of *Shift*'s policies avoids an HDD read upon rejection, significantly reducing the amount of cache fills (96% less than *Dynamic*) compared to CacheLib's baseline policies.

These results show that *Shift*'s threshold tuning mechanism is effective at maximizing IO absorption given a write constraint, without incurring excess *Tectonic* cluster reads.

How effective is reinsertion? We evaluated if reinsertion was effective at reducing HDD reads compared to admission only. We repeated the Synchronized workload with a write limit of 1 GB/s and a reinsertion threshold 1.0 greater than the dynamic admission threshold with a minimum admit of 1.1. We compared a *Hybrid* admission policy with reinsertion against a baseline *Hybrid* policy without reinsertion.

Table 4 shows the hit rate and HDD IO with reinsertion enabled, normalized to the baseline without reinsertion. We observe that enabling reinsertion resulted in similar hit rates (3% increase), while reducing HDD reads by 18%. However, compared to the limited flash write rate of the baseline (≈ 300 MB/s), enabling reinsertion with dynamic threshold tuning resulted in *Shift* always hitting the write limit, since reinsertions caused more reinsertions until the limit was exceeded. Our takeaway is that currently, reinsertions may be effective when reducing HDD reads are prioritized over reducing flash writes. However, potential future optimizations in CacheLib (see Section 7) can harness reinsertion's benefits while eliminating the write overheads caused by continued reinsertions.

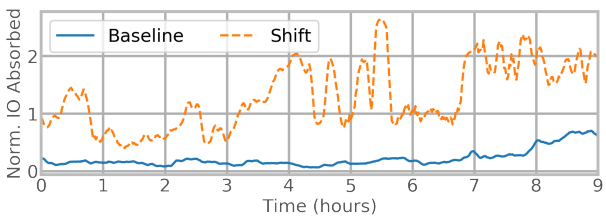


Figure 10: Production results comparing *Shift* to an expert manual-tuned policy that admits only IO-heavy tables.

6.2 Production Results

We have enabled the policies presented in Section 5 in our production clusters. Since the mix of training jobs and resources varies across datacenters, we are continuously tuning policy configurations for each cluster. These policies have helped *Shift* save significant amounts of power.

To demonstrate this, Figure 10 shows a representative trace of the IO absorbed by *Shift* nodes in a petabyte-scale production cluster over the course of 9 hours. We show a baseline that uses an *Expert*-tuned admission policy on top of LRU, which admitted only high-IOPS tables to cache; no admission policy (admit all) showed near-zero IO absorption due to significant cache contention across training jobs. We compared against *Shift* using a *Hybrid* policy, with a minimum admit threshold of 3.0 and without reinsertion. Both used client eligibility policies and our production write rate limit (the baseline additionally used *Dynamic* admission if necessary), and we normalized results to the "power-neutral" IO absorption point: the required amount of IO *Shift* needs to absorb to reduce power consumption compared to using only HDDs.

The *Expert* admission policy is ineffective at saving power due to its inability to capture the limited data reuse of training jobs we characterized in Section 3. Simply deploying a flash cache without intelligent and adaptable cache policies is inefficient; our production trace shows that doing so would only absorb $0.21\times$ the IO needed to achieve power neutrality. By employing the application-aware policies presented in Section 5, we show that *Shift* can exploit the unique characteristics of training jobs, saving 29% of power relative to using only HDDs for training data storage. At our scale, this corresponds to a massive efficiency improvement.

7 Lessons Learned and Open Questions

Define the right interface to users. Our focus on designing a *transparent* but *intelligent* interface to users was instrumental in the success of *Tectonic-Shift*. A transparent interface allowed us to quickly onboard new users by simply configuring the cache eligibility policy, requiring *zero* application modifications. Since applications could be agnostic to their use of *Shift*, this allowed us to dynamically manage the deployment and operation of *Shift* with fine granularity. For example, we could gradually roll-out to a new customer, A/B test different policies across SNs, or even roll-back to reduce request pressure — all without affecting customers' performance.

At the same time, we quickly learned that it was essential to work closely with customers to maximize *Shift* performance. We collaborated heavily with AI infrastructure and ML engineering teams to understand and extract the right set of application metadata to optimize *Shift* policies. Looking forward, we believe that a wide range of other ML domains (e.g., vision, NLP, etc.) and non-ML applications will benefit from *Shift*. Since these applications use the same *Tectonic* API, our focus can simply lie in working with customers to extracting the best features to maximize *Tectonic-Shift* efficiency.

Rely on a slate of robust testing, experimentation, and monitoring mechanisms. We use multiple tools such as stress tests, production A/B testing, trace-based simulation, shadowing, and dashboards to continuously tune the multiple *Shift* policies and configurations discussed in Section 5. These tools have also helped ensure a stable deployment of *Shift*. For example, data corruptions are common at our scale, requiring us to compute, store, and verify checksums for each *Shift* segment. To ensure data correctness prior to deployment, we used a shadow deployment to have clients fetch *Shift* checksums to compare against data read from *Tectonic*.

Effectively use DRAM. CacheLib uses both DRAM and flash to store data. For our use case, DRAM capacity was negligible relative to the orders of magnitude larger flash capacity. Instead, we prioritized using DRAM to store metadata (e.g., the RHW) to improve the effectiveness of *Shift* policies, but we still had to reserve tens of gigabytes of memory for CacheLib to buffer and serve requests at high throughput. Further CacheLib optimizations to reduce memory requirements and an investigation into the optimal split between data and metadata may further improve *Shift* performance.

Can data placement and job routing policies improve cache performance? A key opportunity we foresee is co-designing *Tectonic-Shift* with data placement and training job routing policies. Data placement policies govern how tables are replicated across our datacenters. Cache-aware data placement policies can help reduce cache working set sizes by intelligently reducing data replication while balancing for data availability. Cache-aware job routing policies can improve the IO absorbed by *Shift* by coalescing jobs that read similar data within the same datacenter.

How much can priority-aware evictions improve cache performance? Section 6 showed that while reinsertion was effective, it required significant flash writes due to reinsertion cycles. Reinsertions are necessary because CacheLib only offers LRU or FIFO eviction. We believe that further optimizations in CacheLib, such as allowing selective evictions based on a cache object's priority, can harness the benefits of reinsertion without write overheads.

Can historic and future knowledge inform better cache policies? *Shift* provides an extensible framework to build cache admission and reinsertion policies based historic and

future information. While we demonstrated that a hybrid policy based on the maximum of historic and future priorities was effective, a promising research direction is to explore novel cache policies that can leverage future information. For example, a potential cache policy may account for *when* objects will be read in the future and prioritize earlier objects.

8 Related Work

Software Flash/DRAM Caches. Systems such as Redis [36], memcached [45], RAMCloud [49], and Pocket [29] are widely used as caches across datacenter applications. These software caches commonly manage DRAM and/or flash using a mix of policies including admission (e.g., TinyFLU [12] and LARC [20]) and eviction/replacement policies (e.g., ARC [39], LRU [47], 2Q [22], and OPT [5]) that leverage historical (or oracular) access patterns. Techniques to predict file access patterns, e.g., access trees [33], are also well studied.

Recent works have also proposed mechanisms to dynamically tune these policies using ML models [34, 52, 59], hardware access signatures [63], NLP techniques [17, 69], and other heuristics [40, 66]. Meanwhile, flash-specific policies [6, 13, 38, 56, 65] largely focus on managing flash write amplification (WA). CacheSack [66] is used as the admission policy for Google's Colossus Flash Cache, which shares a similar goal to *Shift* in absorbing IO from HDDs. CacheSack splits objects by category and tunes the admission policy of each category. Janus [4] is also a flash tier used in Google's Colossus file system, but instead requires files to be written to flash first before being evicted to HDDs.

Various existing storage systems also leverage architectures similar to *Tectonic-Shift*. Swift uses a set of configurable hash rings to map objects to their respective storage device [48]. Numerous file systems leverage heterogeneous storage devices to optimize for performance and efficiency across various applications [24, 60, 62, 64]. For example, burst buffers, typically consisting of IO-performant devices such as flash, are commonly used to absorb peaks of high IO-demand from backend (e.g., HDD) storage systems in high-performance computing applications [35].

Tectonic-Shift is a composite storage fabric that employs a mix of cache policies across *Tectonic* Clients and *Shift* Storage Nodes to maximize its efficiency. *Shift* runs a CacheLib [6] instance in each SN. We use comparatively large segments (256 KB) and rely on CacheLib's Large Object Cache to handle WA. Each *Shift* SN dynamically tunes its cache policies, including admission and reinsertion, independent of CacheLib's. While *Tectonic-Shift* shares a similar goal with Google's CacheSack [66] and Janus [4], and leverages well-known techniques such as consistent hashing and heterogeneous devices, it uniquely targets industrial ML training jobs and adopts novel application-aware policies that infer future access patterns from job specifications.

ML-specific Caches. Recent work has shown the utility of caches for ML training workloads. CoorDL [41] eliminates data stalls in single-server training using local SSDs. Quiver [30] and OneAccess [23] cache and share data across highly-synchronized HP tuning jobs. DIESEL [61] targets training workloads over small files (e.g., images). DLFS [71] and DeepIO [70] randomize mini-batches using specialized hardware. Cachew [16] builds on tf.data [44], and puts intermediate data in cloud storage to reduce preprocessing costs.

While other ML caches require adoption effort, *Tectonic-Shift* is completely transparent to users. Furthermore, Section 3 explored why industrial ML training workloads present novel challenges not addressed by these caches. *Tectonic-Shift* is designed for exascale and continuously serves traffic to datacenter-scale GPU training clusters.

Production ML Workload Characterization. tf.data [44] provided an ML training workload characterization at Google, highlighting similar traits such as prevalent data reuse and selective reading. *Tectonic-Shift* is motivated by and builds upon prior characterization of Meta's training workloads [68].

Distributed File Systems. *Tectonic-Shift* is built on top of *Tectonic* [50] and provides the same API and append-only semantics as the *Tectonic* File System. Other distributed file systems, such as Spanner [10], GFS [15], Colossus [19], HDFS [54], and Lustre [53], are used across industry.

9 Conclusion

We presented *Tectonic-Shift*, the composite storage fabric used in Meta's production ML training infrastructure. *Tectonic-Shift* maximizes efficiency by balancing storage and IO capacity across HDD and flash. We provided an in-depth workload characterization and design space exploration which guided the principled design of *Tectonic-Shift*. *Shift* employs a set of application-aware policies that infer and exploit future access patterns using job specifications. We demonstrated how *Shift* absorbed $1.51-3.28\times$ more IO than an LRU flash cache and improved the power efficiency of *Tectonic-Shift* by 29%.

Acknowledgments

We are grateful to the anonymous reviewers and to our shepherd, Apoorve Mohan, whose comments have greatly helped improve this paper. We would also like to acknowledge the contributions of Rocky Wang, Mario Consuegra, Cem Cayiroglu, Jolene Tan, and many others at Meta who have played a vital role in this endeavor. Christos Kozyrakis was partially supported by the Stanford Platform Lab and its affiliates, and by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. Mark Zhao was supported by a Stanford Graduate Fellowship while at Stanford University.

References

[1] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, Janet L. Wiener, and Okay Zed. Scuba: Diving into data at facebook. *Proc. VLDB Endow.*, 6(11):1057–1067, aug 2013.

[2] B. Acun, M. Murphy, X. Wang, J. Nie, C. Wu, and K. Hazelwood. Understanding training efficiency of deep learning recommendation models at scale. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 802–814, Los Alamitos, CA, USA, mar 2021. IEEE Computer Society.

[3] Jason Adrian. Introducing bryce canyon: Our next-generation storage platform. https://engineering.fb.com/2017/03/08/data-center-engineering/introducing-bryce-canyon-our-next-generation-storage-platform/, 2017.

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

[5] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[6] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768. USENIX Association, November 2020.

[7] Daniel S. Berger, Ramesh K. Sitaraman, and Mor Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 483–498, USA, 2017. USENIX Association.

[8] Matt Bowman, Abe Garcia, Jun Shen, Haken Michael, Wei Zhang, and Ross Stenfort. Yosemite v3: Sierra point e1.s 2ou flash blade and expansion board design specification. https://www.opencompute.org/documents/e1s-expansion-2ou-1s-server-design-specification-pdf, 2021.

[9] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's distributed data store for the social graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, San Jose, CA, June 2013. USENIX Association.

[10] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264, Hollywood, CA, October 2012. USENIX Association.

[11] Western Digital. Wd gold ssd product brief. https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/product/internal-drives/wd-gold-ssd/product-brief-wd-gold-enterprise-class-nvme-ssd.pdf, 2022.

[12] Gil Einziger and Roy Friedman. Tinylfu: A highly efficient cache admission policy. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 146–153, 2014.

[13] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 65–78, Boston, MA, February 2019. USENIX Association.

[14] Apache Software Foundation. Apache orc: High-performance columnar storage for hadoop. https://orc.apache.org/, 2022.

[15] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 29–43, New York, NY, USA, 2003. Association for Computing Machinery.

[16] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A. Thekkath, and Ana Klimovic. Cachew: Machine learning input data processing as a service. In *2022 USENIX Annual Technical Conference*

(USENIX ATC 22), pages 689–706, Carlsbad, CA, July 2022. USENIX Association.

[17] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. Learning memory access patterns. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1919–1928. PMLR, 10–15 Jul 2018.

[18] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629, 2018.

[19] Dean Hildebrand and Denis Serenyi. Colossus under the hood: a peek into google's scalable storage system. https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system/, April 2021.

[20] Sai Huang, Qingsong Wei, Jianxi Chen, Cheng Chen, and Dan Feng. Improving flash-based disk cache with lazy adaptive replacement. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2013.

[21] Facebook Inc. Hive-dwrf. https://github.com/facebookarchive/hive-dwrf, 2015.

[22] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, page 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[23] Aarati Kakaraparthy, Abhay Venkatesh, Amar Phanishayee, and Shivaram Venkataraman. The case for unifying data loading in machine learning clusters. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[24] Elena Kakoulli and Herodotos Herodotou. Octopusfs: A distributed file system with tiered storage management. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 65–78, New York, NY, USA, 2017. Association for Computing Machinery.

[25] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, page 654–663, New York, NY, USA, 1997. Association for Computing Machinery.

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

[27] Andrej Karpathy. Software 2.0. https://karpathy.medium.com/software-2-0-a64152b37c35, Mar 2021.

[28] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[29] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association.

[30] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An informed storage cache for deep learning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 283–296, Santa Clara, CA, February 2020. USENIX Association.

[31] Frederic Lardinois. Google launches a 9 exaflop cluster of cloud TPU v4 pods into public preview. https://techcrunch.com/2022/05/11/google-launches-a-9-exaflop-cluster-of-cloud-tpu-v4-pods-into-public-preview/, May 2022.

[32] Gyewon Lee, Irene Lee, Hyeonmin Ha, Kyunggeun Lee, Hwarim Hyun, Ahnjae Shin, and Byung-Gon Chun. Refurbish your training data: Reusing partially augmented samples for faster deep neural network training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 537–550. USENIX Association, July 2021.

[33] Hui Lei and Dan Duchamp. An analytical approach to file prefetching. In *USENIX 1997 Annual Technical Conference (USENIX ATC 97)*, Anaheim, CA, January 1997. USENIX Association.

[34] Evan Zheran Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. An imitation learning approach for cache replacement. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org, 2020.

[35] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. On the role of burst buffers in leadership-class storage systems. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11, 2012.

[36] Redis Ltd. Redis. https://redis.io/, 2022.

[37] Sarang Masti. How we built a general purpose key value store for facebook with zippydb. https://engineering.fb.com/2021/08/06/core-data/zippydb/, Aug 2021.

[38] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 243–262, New York, NY, USA, 2021. Association for Computing Machinery.

[39] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, San Francisco, CA, March 2003. USENIX Association.

[40] Michael P. Mesnier and Jason B. Akers. Differentiated storage services. *SIGOPS Oper. Syst. Rev.*, 45(1):45–53, feb 2011.

[41] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in dnn training. *Proc. VLDB Endow.*, 14(5):771–784, jan 2021.

[42] 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. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 993–1011, New York, NY, USA, 2022. Association for Computing Machinery.

[43] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. f4: Facebook's warm BLOB storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 383–398, Broomfield, CO, October 2014. USENIX Association.

[44] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. Tf.data: A machine learning data processing framework. *Proc. VLDB Endow.*, 14(12):2945–2958, jul 2021.

[45] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, April 2013. USENIX Association.

[46] Kunle Olukotun. Designing computer systems for software 2.0. https://iscaconf.org/isca2018/docs/Kunle-ISCA-Keynote-2018.pdf, June 2018.

[47] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, page 297–306, New York, NY, USA, 1993. Association for Computing Machinery.

[48] Openstack. The rings. https://docs.openstack.org/swift/latest/overview_ring.html, 2023.

[49] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3), aug 2015.

[50] 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. Facebook's tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 217–231. USENIX Association, February 2021.

[51] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[52] Liana V Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning cache replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 341–354, 2021.

[53] Philip Schwan et al. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, volume 2003, pages 380–386, 2003.

[54] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[55] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544, Santa Clara, CA, February 2020. USENIX Association.

[56] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: Advanced photo caching on flash for facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 373–386, Santa Clara, CA, February 2015. USENIX Association.

[57] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, August 2009.

[58] Suketu Vakharia, Peng Li, Weiran Liu, and Sundaram Narayanan. Shared foundations: Modernizing meta's data lakehouse. In *The Conference on Innovative Data Systems Research, CIDR*, 2023.

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

[60] Lukas Vogel, Viktor Leis, Alexander van Renen, Thomas Neumann, Satoshi Imamura, and Alfons Kemper. Mosaic: A budget-conscious storage engine for relational database systems. *Proc. VLDB Endow.*, 13(12):2662–2675, jul 2020.

[61] Lipeng Wang, Songgao Ye, Baichen Yang, Youyou Lu, Hequan Zhang, Shengen Yan, and Qiong Luo. Diesel: A dataset-based distributed storage and caching system for large-scale deep learning training. In *49th International Conference on Parallel Processing - ICPP*, ICPP '20, New York, NY, USA, 2020. Association for Computing Machinery.

[62] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The hp autoraid hierarchical storage system. *ACM Trans. Comput. Syst.*, 14(1):108–136, feb 1996.

[63] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, and Joel Emer. Ship: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, page 430–441, New York, NY, USA, 2011. Association for Computing Machinery.

[64] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. Autoscaling tiered cloud storage in anna. *Proc. VLDB Endow.*, 12(6):624–638, feb 2019.

[65] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, Swaminathan Sundararaman, and Robert Wood. Hec: Improving endurance of high performance flash-based cache devices. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, New York, NY, USA, 2013. Association for Computing Machinery.

[66] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister. CacheSack: Admission optimization for google datacenter flash caches. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1021–1036, Carlsbad, CA, July 2022. USENIX Association.

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

[68] 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, and Parik Pol. Understanding data storage and ingestion for large-scale deep recommendation model training: Industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 1042–1057, New York, NY, USA, 2022. Association for Computing Machinery.

[69] Giulio Zhou and Martin Maas. Learning on distributed traces for data center storage systems. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 350–364, 2021.

[70] Y. Zhu, F. Chowdhury, H. Fu, A. Moody, K. Mohror, K. Sato, and W. Yu. 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)*, pages 145–156, 2018.

[71] Yue Zhu, Weikuan Yu, Bing Jiao, Kathryn Mohror, Adam Moody, and Fahim Chowdhury. Efficient user-level storage disaggregation for deep learning. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12, 2019.