cedar: Optimized and Unified Machine Learning Input Data Pipelines

Mark Zhao
Stanford University
myzhao@cs.stanford.edu

Emanuel Adamiak
Stanford University
adamiak@stanford.edu

Christos Kozyrakis
Stanford University
christos@cs.stanford.edu

ABSTRACT

The input data pipeline is an essential component of each machine learning (ML) training job. It is responsible for reading massive amounts of training data, processing batches of samples using complex transformations, and loading them onto training nodes at low latency and high throughput. Performant input data systems are becoming increasingly critical due to skyrocketing data volumes and training throughput demands. Unfortunately, current input data systems cannot fully leverage key performance optimizations, resulting in hugely inefficient infrastructures that require significant resources – or worse – underutilize expensive accelerators.

To address these demands, we present *cedar*, an optimized and unified programming framework for ML input data pipelines. *cedar* allows users to define a training job's data pipeline using composable operators that support arbitrary ML frameworks and libraries. *cedar*'s extensible optimizer systematically combines and applies performance optimizations to the pipeline. *cedar* then orchestrates pipeline processing across configurable local and distributed compute resources to efficiently meet the training job's data throughput demands. Across eight pipelines, *cedar* improves performance by up to 1.87× to 10.65× compared to state-of-the-art input data systems.

PVLDB Reference Format:
Mark Zhao, Emanuel Adamiak, and Christos Kozyrakis. *cedar*: Optimized and Unified Machine Learning Input Data Pipelines. PVLDB, 18(2): 488 - 502, 2024.
doi:10.14778/3705829.3705861

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/stanford-mast/cedar.

1 INTRODUCTION

Every deep machine learning (ML) training job relies on an *input data pipeline* to transform raw datasets into prepared training samples (i.e., mini-batches of tensors), ready to be consumed by the ML framework (e.g., PyTorch [73] or TensorFlow [1]). These pipelines are executed by *input data systems* such as PyTorch's DataLoader [78] or TensorFlow's tf.data [69]. Input data systems are a distinct and essential component of the end-to-end ML data pipeline, complementing the traditional parallel processing frameworks, such as Spark [90] and Beam [2], commonly used for *offline* ML data ingestion. In contrast, input data pipelines require *online* preprocessing using diverse domain- and ML framework-specific tensor operations that vary heavily across training jobs. Input data systems are designed to execute these pipelines while meeting the stringent performance requirements [8, 65, 94] of each training job.

For example, a computer vision (CV) input data pipeline may use various Python libraries and UDFs (e.g., OpenCV [15] or torchvision [79]) to decode JPEG images from a dataset, apply random augmentations such as crops and distortions to each image, and convert batches of images into a tensor. Each training job uses an input data system to continuously execute the input data pipeline, potentially over multiple epochs, throughout the training job's lifetime. The input data system must carefully match its throughput to accelerator demands. This avoids data stalls [65], which degrade training throughput, without over-provisioning input data resources.

Recently, the input data throughput required by training jobs has grown at an immense rate [65, 69, 93, 94], driven by specialized hardware [9, 43, 82], optimized software techniques [34, 35], and massive training clusters [42, 53, 62, 63, 67]. To avoid costly data stalls, companies such as Google [8], Alibaba [93], and Meta [94] have deployed distributed input data services. While simply scaling-out compute addresses performance bottlenecks, it comes with an immense resource cost. Meta's DPP can require dozens of CPU servers to support a *single* GPU server [94], while a single model at Google can require more than five thousand preprocessing workers [8]! To continue scaling ML infrastructure, *it is critical to optimize both the performance and efficiency of input data systems.*

While recent research has explored various performance optimizations such as caching [20, 31, 41, 46, 51, 65, 92, 96], offloading to high-performance backends [31, 50, 69, 87, 93, 94], prefetching [69], and fusion [69], current input data systems are insufficient for several reasons. First, current systems apply optimizations in an *isolated* manner using bespoke and *inextensible* solutions. Combining multiple optimizations is crucial; however these systems cannot navigate the complex search space that is needed to enable this combination. Thus, users are currently forced to pick and choose optimizations, sacrificing performance and efficiency. Secondly, current systems are *not context-aware*, precluding key optimizations that require understanding application semantics. Finally, current systems are *specialized* – they cannot support the wide breadth of ML frameworks, domain-specific libraries, and execution engines in use across the ML training landscape. For example, many input data systems, including tf.data [69], tf.data service [8], Cachew [31], Plumber [50], FastFlow [87], PRESTO [41], and GoldMiner [93] rely on TensorFlow's dataflow graph and execution backend, limiting their compatibility with non-TensorFlow frameworks such as PyTorch. This is especially worrisome given TensorFlow's decline

Proceedings of the VLDB Endowment, Vol. 18, No. 2 ISSN 2150-8097.
doi:10.14778/3705829.3705861

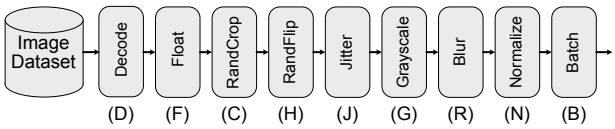

Figure 1: A computer vision input data pipeline applies a sequence of augmentations to each image.

among ML practitioners – *supporting only 2% of recent ML research* as of September 2024 [21].

A Motivating Example. To illustrate these limitations, consider the challenges faced by an ML practitioner training a PyTorch CV model. The practitioner uses the input data pipeline shown in Figure 1 to apply augmentations [18, 19] to each image. Suppose during testing, the practitioner discovers a data stall [65]. The practitioner recalls some optimizations, such as caching, using distributed workers, reordering operators, and fusion, and begins manually testing different execution plans. They quickly realize both the complexities in applying a single optimization (discovering that caching can *harm* throughput), and the difficulties in combining them (resolving an ideal operator fusion that conflicts with an ideal operator ordering)[1]. Upon realizing the billions of execution plans they need to manually implement and test, they decide to port their entire model and pipeline to TensorFlow to leverage its distributed input data frameworks, such as tf.data service [8] or FastFlow [87]. Upon observing that the disaggregated backend needs further optimizations to improve resource efficiency, the practitioner is forced to reluctantly continue manually exploring optimizations.

To address this gap, we believe there is a strong need for an input data framework that can systematically apply a suite of system optimizations and navigate the complex search space that these optimizations introduce. Such a framework must address the unique challenges of ML input data pipelines, which render the straightforward application of traditional query optimizers ineffective. These challenges include the need to support diverse ML frameworks, and the ability to optimize pipelines – programmed via opaque Python UDFs – that heavily rely on domain-specific preprocessing libraries and execution engines. The application of these optimizations must also be *context-aware*, respecting requirements (e.g., randomness) and leveraging novel optimization opportunities (e.g., semantic-preserving reorderings[2]) presented by input data pipelines.

Our Solution. We present *cedar*, an optimized and unified Python-native programming framework for ML input data pipelines. *cedar* transparently enables *systematic*, *context-aware*, and *general* optimizations to meet the high performance and efficiency needs of modern ML training systems. *cedar* allows ML practitioners to easily build input data pipelines by linking together modular native and higher-order operators functionally. These pipelines can support a wide breadth of ML frameworks (e.g., PyTorch and TensorFlow) and domain-specific preprocessing libraries. Meanwhile, *cedar* automatically optimizes and manages the execution of the pipeline to meet the training job's throughput requirements, eliminating data stalls with high resource efficiency. To do so, *cedar* introduces an extensible Optimizer, which systematically applies a combination of state-of-the-art and novel context-aware input data optimizations to improve throughput on a per-resource basis. Importantly, to maintain the semantic correctness of black-box UDFs under these optimizations, the Optimizer leverages a set of simple yet expressive hints specified by the practitioner that provide essential domain knowledge to *cedar*. Then during runtime, *cedar* dynamically orchestrates processing across an extensible and scalable set of execution engines, such as a distributed cluster or local process pools on the training nodes' CPUs, according to the optimized plan. *cedar* continuously monitors and right-sizes resources, efficiently meeting the training job's throughput demands.

We evaluated *cedar* on a diverse set of eight input data pipelines across ML domains, using both local and distributed execution engines. By understanding the complex systems dynamics that arise in each pipeline, *cedar* successfully leverages a mix of optimizations and engines to improve each pipeline's throughput. *cedar* outperforms state-of-the-art ML input data systems, including tf.data [69], tf.data service [8], FastFlow [87], Plumber [50], Ray Data [81], and PyTorch's DataLoader [78] by up to 1.87× to 10.65× while utilizing the same set of resources. *cedar* then effectively scales resource allocations to meet diverse training throughput demands, translating this performance benefit to high input data system efficiency.

In summary, we make the following contributions.

- We introduce an extensible optimization framework for ML input data pipelines that automatically explores the massive combinatorial search space introduced by concurrent optimizations.
- We introduce easy-to-use interfaces that enable novel input data optimizations (i.e., semantic-preserving reorderings) that rely on the domain knowledge of black-box UDFs.
- We present *cedar*, a unified input data framework that transparently optimizes and orchestrates pipeline processing, supporting a wide range of ML frameworks and execution engines.

cedar provides an important, yet missing, foundation for input data systems research, analogous to the influence extensible optimizers (e.g., Spark SQL's Catalyst [7]) and execution interfaces (e.g., Beam's Runners [28]) have had in traditional data processing.

2 ML DATA INGESTION BACKGROUND

ML training jobs rely on data ingestion pipelines to transform raw operational data into structured samples (i.e., tensors) interpretable by ML frameworks such as PyTorch [73] and TensorFlow [1]. As shown in Figure 2, these pipelines traditionally consist of two phases: offline *feature engineering* and online *input data processing*.

Offline Feature Engineering. Feature engineering pipelines validate, aggregate, join, and transform raw operational data into structured *datasets*, off the critical path of training. Since feature engineering predominantly requires traditional extract-transform-load (ETL) tasks, ML practitioners commonly use general-purpose distributed processing frameworks such as Spark [90], Flink [17], and Beam [2]. These ETL tasks are independent from training jobs. They run prior to training and materialize datasets that are stored across a variety of systems, from training nodes' local file systems to distributed data lakes [6, 23, 71, 96].

Online Input Data Pipelines. Each training job concurrently runs an input data pipeline to perform the "last-mile" of processing – extracting a job-specific subset of training samples from stored

[1]We experimentally illustrate these scenarios in Section 3.

[2]e.g., applying blur and crop to an image in either order yields the same semantics.

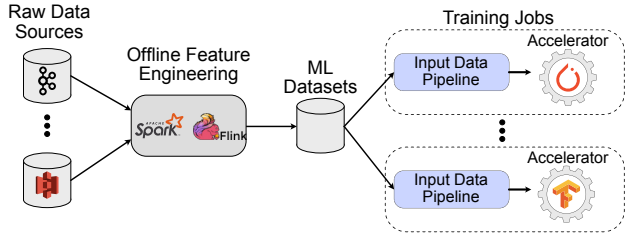

Figure 2: ML training data pipelines consist of an offline feature engineering and an online input data processing stage.

datasets, transforming them into ready-to-use mini-batches of tensors, and loading tensors into the host memory of the training node – all on-the-fly. This "online" processing is needed because operations often vary heavily across jobs or even across epochs within the same job, making materialization highly inefficient. For example, NLP practitioners often experiment with tokenization algorithms [39] across model architectures. Recommendation models require different hashing configurations depending on their embedding table dimensions [83]. Many domains, including CV [22], NLP [26], and speech [72], apply random augmentations that reprocess each sample every epoch to improve model generalization.

As a result, input data pipelines have requirements distinct from traditional ETL tasks, demanding a dedicated class of systems. Input data systems must right-size resources and generate each mini-batch to meet strict low latency ($O(100\mu s - 1ms)$ per step) [8] and high throughput ($O(10GB/s)$ per node) [94] requirements throughout the lifetime of the job, which can take days to weeks. Violations of these requirements can bottleneck expensive accelerators [65]. Furthermore, ML input data pipelines predominantly operate on unstructured tensors within a mini-batch granularity and rely heavily on myriad domain-specific Python libraries and UDFs. For example, practitioners may call a HuggingFace [39] tokenizer or a torchvision [79] crop for each text or image training sample, respectively.

Distributed input data systems, such as Meta's DPP [94] and Google's tf.data service [8], are increasingly deployed to address throughput demands. However, they present an immense resource cost that constrains the scalability of ML training systems. For example, tf.data service deployments at Google commonly use between 2 and 32 (and up to 5K) distributed workers for each training job [8]. Recommendation training jobs at Meta can require dozens of DPP workers for *each* GPU training node – demanding comparable power capacity to the training node itself [94].

3 OPTIMIZING INPUT DATA PIPELINES

3.1 Requirements

Optimizing the performance and efficiency of input data systems is essential. Recent systems have begun to explore techniques such as parallelism and disaggregation [31, 50, 69, 87, 93, 94], caching [20, 31, 41, 46, 51, 54, 65, 92, 96], operator fusion [69, 81], and inter-job coordination [31, 45, 51, 65, 92]. Unfortunately, current input data systems cannot enable the *systematic*, *context-aware*, and *general* optimizations critical to achieving this goal.

Need for Systematic Optimizations. As we deeply explore in Section 3.2, myriad input data optimizations (e.g., offloading, fusion, caching, reordering, and prefetching) can be effective at significantly improving performance. Combining these optimizations is essential, improving throughput by 4.44× versus a single technique in isolation. To do so, systems need a deep understanding of the interactions between optimizations and a systematic approach to navigating the complex tradeoffs and search space introduced.

Current systems cannot systematically combine and apply optimizations. They lack the notion of an extensible optimizer and instead craft bespoke solutions focused only on a limited and isolated set of techniques. For example, Plumber [50] uses a complex linear program to tune operator parallelism, but its formulation is constrained to a single processing node. Meanwhile, FastFlow [87] can leverage distributed workers for processing, but explicitly chooses between only three execution plans, precluding additional optimizations such as caching intermediate outputs or operator fusion. PRESTO [41] introduces a profiler that can inform certain optimizations, such as the optimal location to cache, but requires users to manually reason about and perform these changes.

Need for Context-Aware Optimizations. Unlike traditional data processing applications with standardized APIs, like SQL, input data pipelines heavily rely on opaque Python UDFs with unique operational semantics. This can both complicate and serve as an opportunity for optimizations. For example, caching after certain stochastic operators may harm model convergence [54]. Meanwhile, certain reorderings may yield performance benefits while preserving intended semantics; for example performing a crop before a blur eliminates wasted work. Input data systems require domain knowledge for these context-specific considerations.

Various input data systems such as tf.data [69] and TorchData [76] allow users to build structured pipelines from these UDFs via higher-order operators (e.g., *map* and *filter*). However, these systems still require users to manually reason about and perform optimizations, such as explicitly inserting a *cache* operator in the pipeline. While systems like Cachew [31] can automatically identify optimal caching points, they require users to explicitly indicate where caching is allowed. By requiring users to directly modify the pipeline, such interfaces limit the integration of further optimizations. For example, with reordering, users would need to modify the combinatorial set of orderings with permissible cache locations. Ideally, input data systems should enable users to provide simple yet expressive hints that facilitate context-aware optimizations and ensure correctness, but without precluding further optimizations.

Need for General Optimizations. Finally, ML practitioners use a variety of preprocessing libraries and ML frameworks tailored to their needs. For example, an NLP practitioner may use HuggingFace's Tokenizer library [39], while an ASR practitioner may require MP3 decoding methods from librosa [59]. Furthermore, these libraries often rely on different execution backends, such as Apache Arrow [27] or TensorFlow kernels. These input data pipelines then supply data to various ML training platforms, such as PyTorch [73], Jax [14], TensorFlow [1], MindSpore [64], and others [21]. Input data systems should be able to apply optimizations across diverse domain-specific libraries, execution engines, and ML frameworks.

Unfortunately, many systems restrict their scope to optimizing a subset of use cases. For example, many (if not most) input data systems, including PRESTO [41], Cachew [31], tf.data service [8], tf.data [69], GoldMiner [93], FastFlow [87], and Plumber [50] are

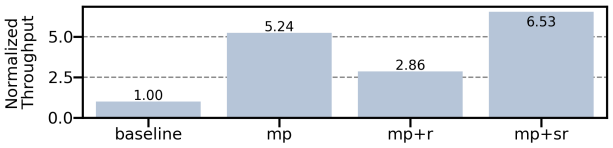

Figure 3: Pipeline throughput, normalized to the baseline, by offloading operators across local and remote backends.

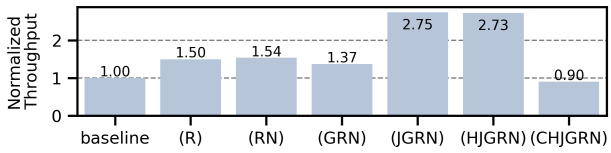

Figure 4: Pipeline throughput, normalized to the baseline, by executing fused operators (see Figure 1) remotely.

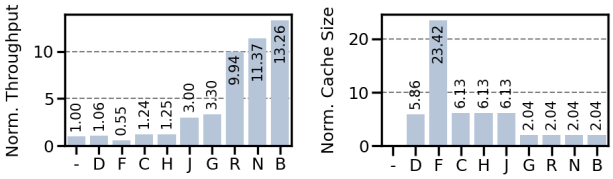

Figure 5: Pipeline throughput and cache size requirement after materializing the output of each operator (see Figure 1), normalized to no caching and the raw dataset size ('-').

built on top of TensorFlow's static graph abstraction. While doing so allows these systems to gain the benefits of TensorFlow's graph optimizations [85], this severely limits their applicability to the limited set of practitioners who rely on TensorFlow [21]. For example, these systems cannot optimize pipelines that rely on third-party training frameworks or libraries. In a similar vein, other recent works use specific execution engines for input data processing (e.g., Ray [66] with Ray Data [81], or GPUs with FusionFlow [47] and DALI [70]). While it is critical to leverage the benefits of these specialized engines, it is also important to not be *limited* to them.

Finally, we note that recent works also address additional considerations, such as multi-tenant environments. For example, Cachew [31], CoorDL [65], Quiver [51], OneAccess [45], and Tectonic-Shift [96] are designed to exploit data reuse in multi-tenant scenarios where datasets and transformations are shared across concurrent training jobs. In this paper, we focus on optimizing the performance and efficiency of input data systems for a single training job. However, because these works leverage input data systems as a foundation (e.g., Cachew builds on tf.data), our optimizations will be essential to the performance of these orthogonal applications.

3.2 The Complex Optimization Space

We begin by analyzing and distilling insights behind an extensive set of optimization techniques that are currently applied in an isolated manner: offloading, fusion, caching, and prefetching. We also introduce a new optimization in the context of input data pipelines: semantic-preserving reorders. We used the representative CV pipeline[3] for SimCLR [18, 19] shown in Figure 1, and we performed experiments on an 8-core (n2-standard-8) and 32-core (n2-standard-32) VM on Google Cloud. The pipeline reads a subset of the ImageNet dataset [25] stored on the file system of the 8-core VM. Unless otherwise specified, each optimization extends a `baseline` that executes all operators in the main data loading process on the 8-core VM. We focus on how these optimizations improve raw input data throughput given a fixed amount of resources, which translates to improved training throughput and efficiency.

Offloading. Many input data systems offload the entire pipeline to parallel execution engines (e.g., thread/process pools [69, 78] or distributed workers [8, 93, 94]). The choice of execution engine presents complex tradeoffs on a pipeline and per-operator basis. Figure 3 shows the throughput of the `CV` pipeline across various engines, normalized to the `baseline`. `mp` offloads execution of the entire pipeline to a local multiprocess pool. `mp+r` offloads the entire pipeline to multiple processes on the remote 32-core VM, with local processes facilitating RPCs. Surprisingly, *more* parallelism *harms* performance because each local process performs more work communicating than it would performing the operators themselves. That being said, remote backends *can* offer benefits if used correctly. `mp+sr` only *selectively* offloads the blur operator, which exhibits a high arithmetic intensity, improving throughput by 25% over `mp`.

Insight. Each operator exhibits diverse performance benefits, *or losses*, across engines. Carefully select *each* operator's engine, as opposed to relying on a single engine for the entire pipeline.

Fusion. Fusing operators can improve performance by eliminating intermediate data transfers. Figure 4 shows the throughput of the `CV` pipeline when fusing and offloading certain operators to the remote 32-core VM. Fusing and offloading neighbors to blur (R) can either improve or *degrade* performance. Fusing grayscale (GRN) harms performance because doing so sends RGB images remotely, incurring I/O costs. Meanwhile, continuing to fuse the compute-intensive jitter (JGRN) is optimal as it best leverages parallelism. Further fusing crop (CHJGRN) eliminates these benefits, as full-sized images now need to be sent to the remote VM.

Insight. Simple heuristics (e.g., fusing all adjacent `maps` [69]) has severe pitfalls. Instead, fusions must be systematically applied based on each operator's dynamics (e.g., I/O and compute demands).

Caching. Caching can improve performance by trading off compute for storage costs. Figure 5 shows the execution time and intermediate cache size observed when caching the output of a specific operator to the disk of the 8-core VM. Caching can harm throughput, such as caching the output of int8 to fp32 conversion (F). Doing so increases the size of each sample, incurring more I/O overheads relative to compute saved. Furthermore, caching after any *random* operator (e.g., crop (C)) would violate the stochastic semantics of the pipeline, harming the model's accuracy. Considering these factors, caching the CV pipeline is largely ineffective without additional optimizations! The best semantic-preserving cache location (after decode (D)) minimally reduces execution time at large storage cost.

Insight. Caching must consider both random operators, and the relative savings in compute compared to storage and I/O costs.

Reordering. While operator reordering is well-known in database systems [48], its benefits have yet to be explored for input data

[3]We refer to each operator by its letter abbreviation in this section.

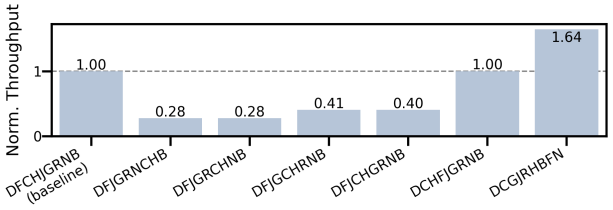

Figure 6: Pipeline throughput (without offloading) by reordering various operators in the CV pipeline (see Figure 1).

pipelines. Figure 6 shows how there is a 5.90× variation in execution time across seven operator orderings. The ideal reordering pushed size-reducing transformations (e.g., crop (C) and grayscale (G)) towards the beginning and size-increasing transformations (e.g., int8 to fp32 conversion (F)) towards the output. Note that certain reorderings can cause minor variations in the output (e.g., reordering a blur and crop) while preserving its overall semantics (a cropped, blurred image). ML training jobs are robust to these variations (e.g., augmentations are often applied in random order [22]).

Insight. Reordering operators based on how they change sample size can improve performance by reducing the compute required for each sample. However, safely reordering operators requires domain knowledge from users to specify permissible reorderings.

Prefetching. Prefetching the output of the pipeline allows input data processing to be overlapped with the training step. For example, we observed a 35% improvement in end-to-end training throughput for the CV pipeline by prefetching its output, assuming a 100*ms* training step. Furthermore, prefetching the output of an offloaded operator can overlap its computation with downstream operators. We observed a 30% improvement in overall input data throughput by prefetching the output of an offloaded blur operator.

Insight. Input data systems should prefetch both the pipeline output and offloaded operators to overlap and pipeline computation.

3.3 Our Approach

Despite their impact, we do not believe that these optimizations are exhaustive; we instead advocate for an extensible platform that can incorporate future optimizations, akin to optimizers in traditional data processing systems. Furthermore, it is critical to *combine* multiple optimizations together to maximize the performance and efficiency of input data systems. As we later show in Figure 11, concurrently applying the above optimizations achieves a 4.44× higher throughput than using a single optimization (local offloading), and a 21.81× higher throughput than the `baseline`. However, combining even the above optimizations requires the exploration of a vast search space: ~ 85 billion plans for the CV pipeline.

To address these challenges and the key requirements in Section 3.1, we argue that input data systems need a higher level of components and abstractions. First, an extensible *optimizer*, leveraging structured cost- and rule-based optimization passes, is needed to systematically explore the complex set of plans generated when combining multiple optimizations. Secondly, an easy-to-use and expressive programming model is essential to support diverse pipelines and to capture the necessary domain knowledge to enable key context-aware optimizations. Finally, well-defined interfaces,

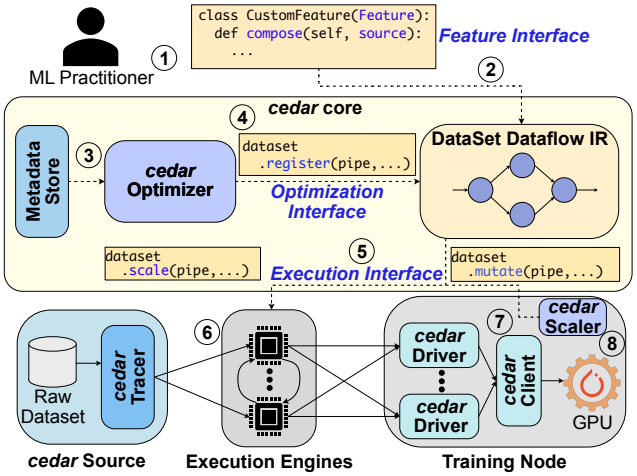

Figure 7: *cedar* block diagram, showing how users can leverage *cedar* to define, optimize, and execute pipelines.

providing an easy-to-manage intermediate representation, is vital to enable future optimizations and execution engines.

4 *CEDAR* FRAMEWORK

To this end, *cedar* introduces higher-level abstractions and components that allow ML practitioners to easily build, optimize, and execute input data pipelines. *cedar* provides native and higher-order operators which practitioners use to define input data pipelines that are *general* – supporting arbitrary ML frameworks and libraries – and *logical* – abstracting away underlying processing details (e.g., which engine or how much parallelism to use). Practitioners can optionally provide a lightweight set of hints to provide domain knowledge for *context-aware* optimizations. *cedar*'s `Optimizer` then statically applies the optimizations presented in Section 3.2 to yield an optimized execution plan that improves performance on a per-resource basis. During runtime, *cedar*'s `Scaler` auto-scales resources to execute this plan, across configurable engines, to efficiently meet training throughput requirements.

Figure 7 shows an end-to-end example. ① A practitioner composes a *logical* input data pipeline (a `Feature`) by functionally chaining together operators (`Pipes`) using the Python Feature API (Section 4.1). ② *cedar* parses the `Feature` into a dataflow graph. ③ *cedar*'s `Optimizer` (Section 5.1) then collects performance statistics for each `Pipe` from past runs or via a short profiling job. ④ Using these statistics, the `Optimizer` applies optimizations via the Optimization interface (Section 4.2) to yield an execution plan.

⑤ Once training begins, *cedar* uses the Execution interface (Section 4.2) to initialize each `Pipe` by running its assigned `Variant` – a *physical* implementation of the operator – on its respective engine. ⑥ Engines continuously process samples using their respective operators. ⑦ Meanwhile, because training jobs ingest data in a data-parallel manner, *cedar* creates a `Client` for each distributed training process (e.g., training node). The `Client` launches `Driver` processes to manage operator processing across each engine and to ingest fully-processed samples into the ML training framework

```
class SimCLRFeature(Feature):
  def compose(self, source):
    ft = source.read_image().fix()
    ft = ft.map(ToFloat).tag("F")
    ft = ft.map(Crop).rand().tag("C")
    ft = ft.map(Flip).rand().depends_on(["C"])
    ft = ft.map(Jitter).rand()
    ft = ft.map(Grayscale)
    ft = ft.map(GaussianBlur)
    ft = ft.map(Normalize).depends_on(["F"])
    return ft.batch(batch_size=...)

source = FileSource(<path_to_imagenet>)
feature = SimCLRFeature()
ds = DataSet(source, feature, backends)
for batch in ds:
    # fit the model to batch
```

Figure 8: *cedar* `Feature` API.

(e.g., PyTorch). ⑧ Finally, the `Client` continuously monitors processing and uses a `Scaler` (Section 5.2) to dynamically right-size the resources allocated to each engine via the Execution interface.

4.1 Feature API

cedar provides an easy-to-use `Feature` API. It allows users to define dataflows (`Features`) by composing together stateless operators (`Pipes`) that implement common data loading primitives (e.g., batching, shuffling) and higher-order functions (e.g., map, filter) that support arbitrary Python UDFs. Each `Pipe` applies a logical transformation to input samples, yielding transformed samples to the downstream `Pipe(s)`. Transformations may be applied one-to-one (e.g., map), many-to-one (e.g., batch), or one-to-many (e.g., reading lines in a file). `Features` may also be non-linear DAGs with a single output node that generates mini-batches of data iteratively. `Pipes` may ingest from/emit to one or more `Pipes` (e.g., zip/unzip).

Users can thus easily define and share `Features`, without needing to manage their underlying execution details. For example, Figure 8 shows a `Feature` for the CV pipeline in Figure 1. Each `Feature` is logical; it is not bound to a specific dataset, nor does it specify *how* the dataflow is executed. Instead, ML engineers define a `Source`, which wraps a raw dataset with a `Pipe`, providing an iterator over raw samples. Users supply one or more `Sources`, the `Feature`, and available engines to *cedar*, which constructs an iterable `DataSet`. Training nodes simply iterate over the `DataSet`, which yields fully processed mini-batches within host memory to be consumed by ML frameworks. Execution is lazy and incremental, allowing *cedar* to scale to large out-of-memory datasets.

Relaxed Operator Dependencies. While the original `Feature` dataflow specifies a *potential* ordering, *cedar* allows users to relax ordering constraints by expressing a dependency graph on top of the original dataflow specification. As shown in Figure 8, if a `Pipe` b depends on a, users can label a with an explicit tag_a (Line 5) and declare the dependency via $b.depends\_on([tag_a])$ (Line 6). Users can also fix the position of a pipe a by calling $a.fix()$, which makes a depend on all upstream `Pipes` and makes all downstream `Pipes` depend on a. As we show in Section 5.1, this allows the `Optimizer` to enumerate semantic-preserving reordered plans.

Table 1: Optimization (top) & Execution (bottom) interfaces.

API	Description
register(pipe)	Registers a new `Pipe` into the `DataSet`.
fuse(pipes)	Registers a new `Pipe` that fuses all input `Pipes`.
update_dfg(graph)	Updates the dataflow graph as specified.
assign(pipe, variant)	Assign a `Variant` to the provided `Pipe`.
set_shards(n)	Assign a number of `Drivers` for each `Client`.
shard(n)	Shard the `Feature` into n `Drivers`.
mutate(pipe, variant)	Mutate the `Pipe` to the specified `Variant`.
scale(pipe, n)	Sets the parallelism of `Pipe` to n.

Random Operators. *cedar* also allows ML engineers to designate which `Pipes` represent random augmentations of data (Lines 5-7), allowing the `Optimizer` to preserve randomness by disallowing caching after random operators.

Correctness and Fault Tolerance. *cedar* ensures that the relaxed operator dependencies and randomness constraints are correctly met. Specifically, if operator B depends on A, *cedar* will never generate a plan where B precedes A. Similarly, if operator C is random, *cedar* will never insert a cache operator D such that C precedes D. Users may choose to not specify dependencies or randomness; *cedar* disables reordering and caching, respectively, to ensure correctness. Inferring dependencies or randomness (e.g., via static code analysis [38, 50]) is left for future work.

cedar also guarantees precise checkpoints and exactly-once semantics, key requirements to ensure that model convergence is not affected by faults. Specifically, each `Source` tags all training samples with a UUID, and the `Client` verifies receipt of all tags (accounting for aggregations and filters). Because `Pipes` are stateless, upon detecting a fault, the `Client` instructs the `Source` to re-emit a specific sample to recompute the result. `Clients` do not return duplicate samples to ensure exactly-once semantics. *cedar* provides a checkpoint API which allows *cedar* to resume processing, skipping received samples, in the event of a training job or `Client` failure.

4.2 Optimization and Execution Interfaces

Optimization Interface. The Optimization interface (Table 1) provides well-defined methods that allow the `Optimizer` to apply multiple optimizations to the `DataSet` in an extensible manner, allowing *cedar* to easily integrate future optimizations. For example, to introduce caching or prefetching, the `Optimizer` creates a cache or prefetch `Pipe`, registers it via `register`, and updates the dataflow via `update_dfg` to insert it at the appropriate location. A fused `Pipe` can be created via `fuse` and inserted into the dataflow, and the dataflow can be reordered via `update_dfg` accordingly.

Execution Interface. The `Optimizer` also specifies pipeline execution via `Drivers` and `Variants`. A `Driver` is an independent Python process that manages the end-to-end processing of a disjoint (data-parallel) subset of samples. Each `Client` can use multiple `Drivers` to parallelize GIL-constrained Python operations within a training node. Meanwhile, a `Variant` is a *physical* implementation of the `Pipe`'s logical operation on a given engine (and potentially for a specific ML framework). For example, a *map(tf.image.resize)* `Pipe` can have `Variants` that executes resize from the tf.image

library a) locally in the Driver's Python process, b) on a distributed worker, or even c) by using the runtime of another framework such as tf.data. Native operators (e.g., batch) can further have specific Variants that process ML framework-specific tensors on a given engine (e.g., PyTorch/distributed or TF/local), which *cedar* selects from based on the ML framework used by the training job.

Once the training job starts, each Client will create Drivers using shard and initiate processing for each Pipe by calling mutate to transform the Pipe to the appropriate Variant. Some Variants allow a configurable amount of parallelism (e.g., process pool size); scale sets the amount of parallelism appropriately. Throughout training, each Client ingests data and manages execution only for its respective training process (e.g., training node), allowing *cedar* to support large-scale distributed training in a decentralized manner. Specifically, each Client will use a local Scaler (Section 5.2) to tune both the parallelism and Variant of each Pipe (via scale and mutate) to meet its training process's throughput requirements.

5 OPTIMIZATION AND DYNAMIC SCALING

cedar first applies global static optimizations to increase the per-resource throughput. During runtime, each Client then dynamically right-sizes the resources used by its Variants to efficiently meet the throughput demanded by its training process.

Tracing and Profiling. The Optimizer relies on a collection of performance statistics within the Metadata Store (Figure 7) to calculate cost models across plans. These statistics are automatically collected by *cedar*, which traces the execution of each Pipe. Each Source periodically marks emitted samples to be traced. Each Pipe transparently tags traced samples with statistics, including the sample's execution latency and size (bytes), and the Pipe's current Variant and prefetch buffer length (if applicable). Upon reception, the Client updates the Metadata Store with traced results and a measure of the current overall throughput.

The Optimizer requires a core set of statistics. These include the throughput $tput_{base}$ of the baseline plan, G_{base}, which executes the un-optimized pipeline *locally* (i.e., within a single Driver Python process). For each Pipe p, the Optimizer also requires the average latency to process a sample, $lat_{base}(p)$, and its average input and output sample sizes, $size_{in}(p)$ and $size_{out}(p)$, respectively. Finally, for each Variant v available for p, the Optimizer requires the average DataSet throughput achieved by offloading only p to v, $tput_v(p)$. If statistics are insufficient (e.g., from a previous job) or performance characteristics are different (e.g., the infrastructure changes), the Optimizer runs a short profiling job.

Profiling and optimization introduce negligible overheads (seconds) compared to long-running (hours-days) training jobs.

5.1 Static Optimization

To explore the optimization search space, the Optimizer iteratively applies a set of cost- and rule-based optimization passes to the dataflow graph, similar to traditional database query optimizers [29, 30, 38, 48]. Within each pass, the Optimizer begins by enumerating possible plans by applying a specific optimization technique to the current plan(s) (e.g., enumerating all possible operator orderings for reordering). The Optimizer then evaluates the enumerated plans using a cost model or a set of rules. To efficiently search the optimization space, each pass prunes plans based on cost and satisfiability (i.e., obeying user-specified dependencies and randomness constraints). By default, each pass outputs the lowest-cost, permissible plan to the next pass. Each optimization pass thus enumerates plans $\mathcal{G} = \{G_1, \cdots, G_n\}$ and determines the cost of a plan G by calculating a $cost(p)$ for each Pipe $p \in G$. It aims to find:

$$G^* = \underset{G \in \mathcal{G}}{\text{argmin}} \sum_{p \in G} cost(p), \quad \text{s.t. } G \text{ satisfies user constraints} \tag{1}$$

A higher cost represents more work and thus lower performance. *cedar* uses a comprehensive cost model that extends $cost(p)$ with each pass, allowing system experts to customize the cost model used by a pass, if needed, in a modular manner.

The Optimizer uses an initial cost model for the profiled baseline plan G_{base}, which weights the cost of each pipe by its fractional latency in the end-to-end pipeline:

$$cost_{base}(p) = \frac{lat_{base}(p)}{\Sigma_{i \in G_{base}} lat_{base}(i)} / tput_{base} \tag{2}$$

We next describe each optimization pass in the order that it is applied. We applied logical passes (e.g., reordering) prior to physical optimizations (e.g., offloading). Within each optimization pass, we present a) how the Optimizer enumerates plans, and b) (if applicable) the cost model used. Using iterative passes allows the Optimizer to be easily extended with further optimizations.

Reordering. The Optimizer first finds the best dataflow ordering.

Enumeration. The Optimizer enumerates all permissible reorderings of the initial plan G_{base}. It does so by removing the output Pipe, o, of G_{base} and recursively calculating the set of all possible reorderings of the shrunk graph. For each shrunk reordering G_s with output Pipe o_s, adding o as a successor to o_s produces a viable reordering. Furthermore, if o and o_s may be reordered according to user constraints, swapping o and o_s also produces a viable reordering (i.e., o precedes o_s). reorder only reorders Pipes within their linear subgraph (i.e., fix()-ing Pipes with > 1 input or output).

Cost model. To calculate the cost of a reordering R, reordering augments the base cost model by calculating a *size scaling factor* $S(p) = size_{out,base}(p)/size_{in,base}(p)$ for each pipe p, representing how it scales the size of its output on average. The cost model then computes the new input size of $p \in R$ as $size_{in,R}(p) = size_{raw} * \Pi_{i \in U} S(i)$, where $size_{raw}$ is the average raw sample size and $U \subset R$ is the ordered sequence of all ancestors of p. The cost model calculates the reordered cost as:

$$cost_R(p) = (size_{in,R}(p)/size_{in,base}(p)) * cost_{base}(p) \tag{3}$$

Thus, reordering assumes that the cost of each Pipe scales linearly with its input sample size (e.g., a tokenizer requires half as much compute to process half as many tokens). However, system experts may customize the cost model for different scaling properties of each pipe (e.g., to scale $cost_{base}(p)$ quadratically with sample size). Furthermore, since $size_{in}$ and $size_{out}$ is the average sample size, reordering optimizes the location of operators that change both selectivity (e.g., filter) and sample size (e.g., crop).

Caching. Next, the Optimizer evaluates if and where to best cache (i.e., materialize) intermediate data.

Enumeration. The Optimizer enumerates plans by creating a new plan for each permissible caching location within the dataflow (i.e., after every operator that does not contain an ancestor that is

marked random). We currently consider only inserting one cache operator in the dataflow (i.e., finding the best cache location).

Cost model. To calculate the cost of a plan with a cache `Pipe` p_{cache}, the cost model simply sets the cost of any exclusive ancestor p of p_{cache} (i.e., all paths from p to the output contain p_{cache}) to zero. To account for I/O costs to read cached data, we calculate

$$cost(p_{cache}) = d * size_{in,R}(p_{cache}) \quad (4)$$

where d is a constant derived from the node's disk I/O throughput, and $size_{in,R}(p_{cache})$ is derived using $S(p)$ as with reordering.

Fusion and Offloading. The `Optimizer` considers offloading and fusion concurrently.

Enumeration. It enumerates all possible offloading plans by generating a set $P_i = \{(p_i, v) | v \in V \text{ and } p_i \text{ supports } v\}$ for each `Pipe` p_i, which contains the set of supported `Variants` for p_i within the user-provided engines V. The `Optimizer` first enumerates plans by taking the Cartesian product between all P_is. Then, the `Optimizer` performs all possible fusions for each plan (e.g., fusing adjacent `Pipes` assigned the same `Variant` if each `Pipe` supports fusion). Furthermore, as mentioned in Section 3.1, caching may preclude the ability to fuse operators. The `Optimizer` thus enumerates plans based on the input plan with and without the inserted cache `Pipe`.

Cost model. To compare costs between plans, the cost model uses Amdahl's Law to determine the benefit that p gains if it is offloaded to a `Variant` v. It assigns a lower cost $cost_v(p)$ to `Variants` that achieve a higher throughput. The cost model uses the overall speedup $s_v(p) = tput_v(p)/tput_{base}$ of v. It then solves for the speedup of p on v and linearly scales the reordered $cost_R(p)$ accordingly. We constrain $cost_v(p) \geq 0$.

$$cost_v(p) = cost_R(p) * \frac{s_v(p)^{-1} - (1 - f(p))}{f(p)} \quad (5)$$

Fusion reduces I/O costs between `Pipes`. To calculate the cost of a fused `Pipe` p, which fuses pipes $q_1, ..., q_n$, the cost model calculates the reduction of I/O relative to the un-fused baseline as $io(p) = (size_{in,R}(q_1) + size_{out,R}(q_n))/(size_{in,R}(q_1) + 2 * size_{in,R}(q_2) + ... + 2 * size_{in,R}(q_n) + size_{out,R}(q_n))$. The cost model then discounts the aggregate costs of all original `Pipes` by the relative I/O savings:

$$cost_{fused,v}(p) = io(p) * (cost_v(q_1) + \cdots + cost_v(q_n)) \quad (6)$$

Prefetching and Sharding. Finally, the `Optimizer` applies a set of rules to prefetch and shard the `DataSet`. It inserts a prefetching `Pipe` after each offloaded (non-*base*) `Variant`, as well as at the end of the dataflow, to allow pipelined execution throughout the input data pipeline. The `Optimizer` also calculates the ideal number of shards (i.e., `Drivers`) to use for each `Client`. To do so, it estimates the throughput of each `Driver` using the cost model. If the throughput is over a threshold, the `Optimizer` runs a single `Driver` *within* each `Client` process to avoid introducing an inter-process communication bottleneck. Otherwise, the `Optimizer` further replicates `Drivers` to improve local parallelism (i.e., one per host CPU core).

Summary. The `Optimizer` applies complex optimizations by enumerating and pruning plans in discrete passes (like query optimizers), allowing *cedar* to search the optimization space in an efficient and extensible manner. Each cost-based optimization pass leverages an informed cost model, based on well-founded principles (e.g., Amdahl's Law), that estimates the cost of each `Pipe` using profiled statistics. Some models apply heuristics to support black-box UDFs; for example, reordering assumes that operator costs scale linearly with input size. However, users may easily customize the model in a modular manner for each `Pipe` if needed, for example by modifying Equation 3 to scale quadratically w.r.t. input size.

5.2 Dynamic Scaling

The `Optimizer`'s static pass is designed to select a high-throughput plan in order to increase the utilization of training accelerators. Since `Clients` independently process samples for its respective training process, each `Client` runs an `Scaler` during training to right-size resources to efficiently meet its training process's throughput demands. The `Scaler` continuously monitors performance, identifies the bottleneck `Pipe`, and tunes its parallelism.

As mentioned above, each `Client` continuously traces and reports runtime metrics during training. To identify the bottleneck, the `Scaler` examines the prefetch buffer at the pipeline output. If the buffer length is over a configurable threshold, the input data pipeline is not the bottleneck. In this case, the `Scaler` selects a random `Pipe` p with a non-*base* `Variant` and scales down its parallelism by a unit (e.g., one process). Importantly, if the current parallelism of p cannot be further decreased, *cedar* will mutate p into the *base* `Variant` to avoid over-provisioning resources.

However, if the output buffer is below a threshold, a bottleneck exists. The `Scaler` will attempt to scale-up the parallelism for the bottleneck `Pipe` p_b. It identifies p_b by examining the set of all `Pipes` which were statically assigned a non-*base* `Variant`, P^*, which represents `Pipes` that benefit from offloading. First, the `Scaler` examines the prefetch buffer of all $p \in P^*$ that are currently offloaded (i.e., non-*base*), selecting the p with the smallest buffer below a threshold. If no such `Pipes` exist (e.g., if all $p \in P^*$ are mutated to the *base* `Variant`), the `Scaler` examines all $p \in P^*$ with *base* `Variants` and selects the p with the largest $lat_{base}(p)$ – the largest speedup opportunity. Given p_b, the `Scaler` will iteratively increase its parallelism by a unit until throughput plateaus, potentially mutating p_b back into a non-*base* `Variant`. If the backend's resources are exhausted, the `Scaler` will scale down another random `Pipe` with the same `Variant` as p_b. The `Scaler` periodically runs (e.g., every minute), scaling resources to meet throughput demands.

The `Optimizer`'s static passes (Section 5.1) are responsible for exploring the complex optimization space presented in Section 3.2. Meanwhile, the `Scaler` is responsible for scaling the resources used to execute the optimized execution plan in order to meet throughput demands. While alternative methods such as Bayesian Optimization [55] or simulated annealing [44] could be applied to scaling, we find that the `Scaler`'s hill-climbing approach is a simple and effective solution. This is because as we increase the parallelism (and thus throughput) of a `Pipe`, we can apply Amdahl's Law [4] to model the throughput of the entire pipeline (as in Section 5.1), resulting in a concave function with respect to parallelism. A similar concave speedup function has long been used to scale parallelism for cases ranging from multicore processors [36, 88] to datacenter workloads [12, 91]. We evaluate the `Scaler` in Section 6.2, and Figure 10 experimentally validates this model.

Table 2: Description of pipelines used to evaluate *cedar*.

Pipeline	Description	Model
CV-{torch,tf}	Decode → Float → RandCrop → RandFlip → Jitter → Grayscale → Blur → Normalize	SimCLR [18, 19]
SSD-{torch,tf}	Decode → Resized Bounding Box Crop → Flip → Distort → Normalize	SSD [57]
NLP-{torch,hf-tf,tf}	Read → Tokenize → Truncate → Embedding	LSTM [37]
ASR	Decode → Resample → Spectrogram → Stretch → Time Mask → Freq. Mask → Mel Scale	RNN-T [33]

6 EVALUATION

We designed *cedar* to support the numerous libraries and frameworks currently used across ML deployments. Since ML practitioners predominantly rely on Python, *cedar* is built from the ground up in ~12K lines of Python. *cedar* supports all popular ML frameworks (e.g., PyTorch, JAX, and TensorFlow) and can execute operations from arbitrary preprocessing libraries that provide a Python API.

Workloads. We evaluated *cedar* on a diverse set of eight ML input data pipelines across computer vision, natural language, and speech domains as shown in Table 2. `CV-torch` and `CV-tf` implemented the SimCLR [18, 19] pipeline in Figure 1 using PyTorch and TensorFlow (TF) operators, respectively. They processed the ImageNet dataset [25] and used the semantic constraints shown in Figure 8. `SSD-torch` and `SSD-tf` implemented the SSD pipeline from the MLPerf Training benchmark [58] using PyTorch and TF operators, respectively. `SSD` pipelines used the COCO dataset [56]. All `SSD` operators were random; Distort was able to be reordered between Decode and Normalize. `NLP-torch`, `NLP-hf`, and `NLP-tf` implemented a standard pipeline for natural language tasks [16, 80]. `NLP-torch` used torchtext [77] operators, while `NLP-hf-tf` and `NLP-tf` used tf.text [86] operators with a Hugging Face [39] and TF tokenizer, respectively. All `NLP` pipelines used the WikiText-103 dataset [61], and all operators were not random and not reorderable. Finally, `ASR` implemented the SpecAugment speech recognition pipeline [72] using the third-party librosa [59] library. `ASR` used the Common Voice dataset [5]. All `ASR` operators were fixed except for Stretch, TimeMask, and FreqMask; TimeMask and FreqMask were random.

6.1 *cedar* Optimizer

We begin by evaluating how effectively *cedar*'s static optimizations improve per-resource throughput compared to state-of-the-art input data systems. As highlighted by recent work [41, 65], end-to-end training throughput T_{e2e} is the minimum of the input data throughput T_p and the GPU computation rate T_g. T_g depends on the model and training infrastructure (e.g., GPU version and networking hardware). The primary goal of an input data optimizer is to increase per-resource throughput (e.g., T_p per CPU core). If $T_p < T_g$ given fixed input data resources (e.g., the training node's host CPU), increasing the per-resource throughput, and thus T_p, directly improves T_{e2e}. Alternatively, if $T_p > T_g$, increased per-resource throughput allows a dynamic scaler (which we evaluate in Section 6.2) to reduce the amount of allocated resources, improving the resource efficiency of input data systems.

Thus, to directly evaluate the per-resource throughput, we evaluated the maximum T_p achieved by systems on two hardware setups. Since training jobs often use CPU host resources for input data processing, we first used a `local` setup where all systems performed processing on a single 8-core VM (`n2-standard-8` on Google Cloud). We provided *cedar* with a Python multiprocessing engine, as well as two engines that used a tf.data or a Ray Data runtime to execute operators, respectively. In this setup, we compared against tf.data [69], Plumber [50], Ray Data [81], and the PyTorch DataLoader [78]. We also used a distributed (`remote`) setup, which provided each system with a remote 32-core VM (`n2-standard-32`) in addition to the `local` VM. In addition to the `local` engines, we provided *cedar* with a `distributed` engine which executed operators on the remote VM. In the `remote` setup, we compared against tf.data service [8], FastFlow [87], and Ray Data [81].

We configured each system to maximize throughput (e.g., enabled auto-tuning in tf.data). We also disabled caching in order to directly evaluate the improvements to computational efficiency (e.g., avoiding degenerate cases where the pipeline output is cached). We explicitly evaluate caching in Section 6.2. Figure 9 shows the input data throughput (T_p) for each system across the eight pipelines. To highlight the importance of input data optimizations on T_{e2e}, we also report the T_g of the representative model for each pipeline shown in Table 2 on an A100 GPU. Because input data pipelines are data-parallel (i.e., independent across training processes), these results directly scale to multi-GPU training environments as each GPU would require a similar input data demand.

***cedar* optimizes local input data processing throughput.** For the `CV` and `SSD` pipelines, *cedar* reordered operators to reduce the overall amount of computation required per sample, which we further explore in Section 6.3. This allowed *cedar* to largely outperform PyTorch, tf.data, Plumber, and Ray Data. For `CV-tf`, tf.data and Plumber were able to achieve a comparable performance due to their underlying TF graph optimizer and performant C++ backend.

For `NLP-torch`, both PyTorch and Ray Data suffered from a serialization bottleneck in sending large embeddings between processes. *cedar* was able to avoid this by fusing Read, Tokenize, and Truncate into one multiprocess `Pipe`, while performing Embedding within the main `Client` process. For `NLP-hf-tf`, tf.data could not optimize the non-TF Tokenizer, while Ray Data achieved slightly higher performance (1.17×) than *cedar* because its Arrow core eliminated intermediate data copies. *cedar*'s extensibility allows it to adopt Arrow as a future `Variant`. For `NLP-tf`, tf.data was able to compile the pipeline into an optimized TF graph. *cedar* recognized this benefit and offloaded execution to the tf.data engine, *without requiring users to modify the pipeline*, allowing *cedar* to near tf.data's throughput (0.91× due to tracing) and out-perform Ray Data.

For `ASR`, *cedar* first reordered Stretch (to decrease work for the Mask operators) and then offloaded execution to the Ray Data engine, obtaining its zero-copy benefits and matching Ray Data's performance. Meanwhile, PyTorch and tf.data suffered from copy overheads, and tf.data could not optimize the non-TF operators.

These results highlight the impact of *cedar*'s extensible optimizer – optimizing the dataflow via reordering and fusion, and offloading and prefetching execution to the best engine – all without user input. *cedar* out-performed tf.data, Plumber, Ray Data, and PyTorch

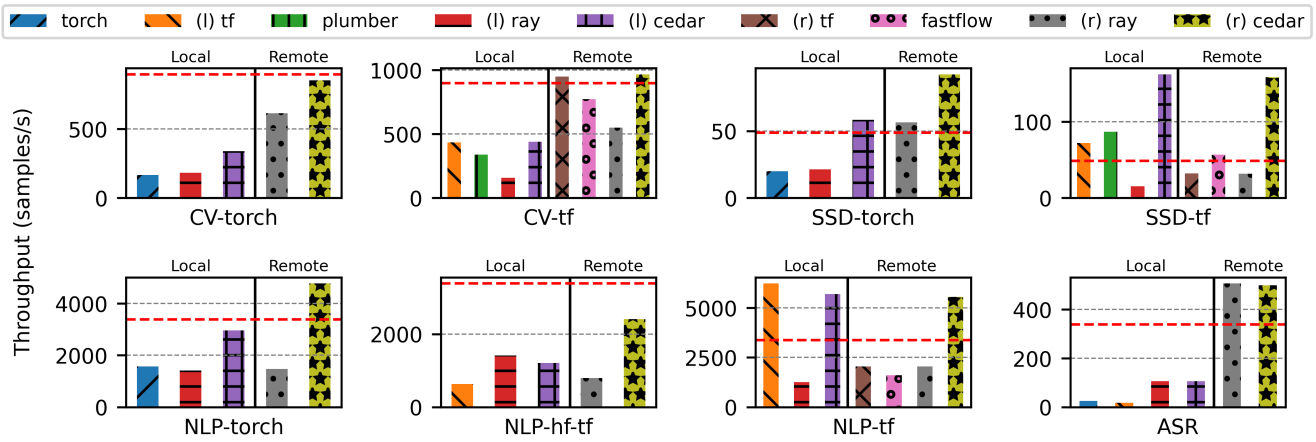

Figure 9: Achieved processing throughput (T_p, higher is better) across eight pipelines. (l) and (r) denote the `local` and `remote` setups of each system, where applicable. The dashed red line marks the GPU computation rate (T_g) for the corresponding model of each pipeline shown in Table 2 on an NVIDIA A100 GPU. Incompatible system/pipelines pairs are not shown.

by up to 6.14×, 1.87×, 10.65×, and 4.28× on the `local` setup, respectively. In almost all cases, the input data throughput T_p achieved by baselines was less than the GPU rate T_g. Since $T_{e2e} = min(T_p, T_g)$, *cedar*'s ability to improve T_p over the baselines directly translates to a corresponding improvement in end-to-end training performance.

***cedar* intelligently uses distributed processing engines.** For the `CV` and `SSD` pipelines, *cedar* generated a similar reordering the `local` setup. *cedar* also offloaded compute-intensive operators (e.g., Distort for `SSD` and Jitter/Blur for `CV`) to the `distributed` engine for `SSD-torch` and the `CV` pipelines, further improving its throughput over the `local` case. Interestingly, *cedar* did not use the `distributed` engine for `SSD-tf`. It instead correctly recognized that offloading operators remotely would incur a slowdown due to data movement overheads. *cedar* improved throughput over tf.data service, FastFlow, and Ray Data; tf.data service matched *cedar*'s performance for `CV-tf` due to its TF graph optimizations.

For `NLP-torch` and `NLP-hf-tf`, *cedar* fused and offloaded only the Read, Tokenize, and Truncate operators to the `distributed` engine, avoiding embedding serialization overheads as in the `local` case. *cedar* correctly decided to not offload any `NLP-tf` operators to the remote VM, avoiding network overheads similar to `SSD-tf`. Meanwhile, tf.data service and Ray Data were both limited by these communication overheads. FastFlow executed processing locally, but inserted logic that hampered TensorFlow's graph compiler. Finally, *cedar* automatically determined that reordering and offloading `ASR` to a distributed Ray Data engine was ideal for the same zero-copy benefits as the `local` setup, matching Ray Data.

cedar was able to effectively and judiciously leverage distributed processing engines, out-performing Ray Data, FastFlow, and tf.data service by up to 4.99×, 3.45×, and 4.94×, respectively. By improving the achievable T_p compared to baselines, given the same remote VM, *cedar* reduces the substantial input data resources needed to meet GPU demands. For instance, linearly scaling the Ray Data's T_p per core to match the T_g demand for `CV-torch` would require remote CPU cores equal in cost to 83% *of the A100 GPU VM* itself (even discounting the cost of the local VM), based on Google Cloud billing rates. Meanwhile, *cedar* reduces this cost to 56%; the `Scaler` (which we evaluate next) can use fewer resources to match T_g demands.

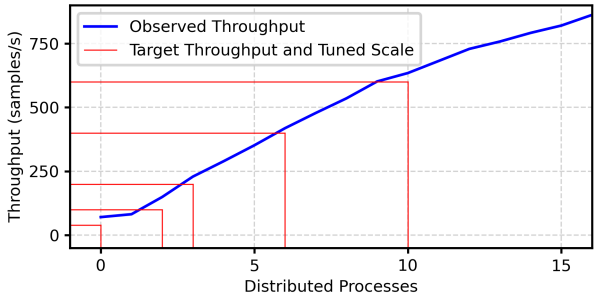

Figure 10: Throughput as the number of distributed processes increases for the `CV-torch` pipeline. The red box shows the scale found by the `Scaler` given a target throughput.

Finally, ***cedar* supports and optimizes diverse input data pipelines.** In contrast, many input data systems could not support all of the evaluation pipelines. DataLoaders and tf.data were limited to PyTorch and TF pipelines, respectively. Systems reliant on TF graphs – tf.data service, Plumber, and FastFlow – could not use any non-TF operator, such as a Hugging Face tokenizer or librosa. Plumber could also not support operators that required non-serializable assets, such as the TF tokenizer in `NLP-tf`.

6.2 Dynamic Scaling and Caching

***cedar* adjusts parallelism to efficiently meet diverse training throughput demands.** While *cedar*'s `Optimizer` successfully improves per-resource throughput, its `Scaler` is responsible for translating this high performance to high resource efficiency by right-sizing resources to meet a given throughput demand (i.e., matching T_p to T_g). To evaluate this, we used the `CV-torch` pipeline with the `remote` setup (other pipelines showed similar results), which *cedar* optimized by fusing and offloading compute-intensive operators (Jitter and Blur) to the `distributed` engine.

Table 3: Auto-cached throughput (normalized to `remote` *cedar*, higher is better) and throughput of the next best cache location across three torch pipelines.

Pipeline	Norm. Throughput with Caching	Next Best Throughput
CV-torch	1.74	1.23
NLP-torch	1.42	1.37
ASR	1.00 (Do Not Cache)	0.82

The blue line in Figure 10 shows the throughput (i.e., T_p) achieved by *cedar* as we swept the number of `distributed` processes, up to saturation. We next set 5 target training throughputs (i.e., T_g), shown by the horizontal red lines. We allowed the `Scaler` to adjust the number of processes; the tuned scale for each target is shown by the intersecting vertical red line. An efficient input data system should *tune parallelism such that this intersection is close to, but below the blue line*. This means that the system provisions the minimal amount of resources to meet demand (i.e., $T_p > T_g$).

Figure 10 shows that *cedar* can not only scale across a wide range of throughput demands, but also efficiently right-size resources; it selected the smallest amount of parallelism to meet each target. Furthermore, *cedar*'s ability to *dynamically* mutate a `Pipe`'s `Variant` even completely deallocated the remote VM in the case of low training throughput, as shown by the zero processes result in Figure 10. This is in contrast to systems such as tf.data service [8] and Cachew [31] which must *always* use distributed processing.

***cedar* optimizes *if* and *where* to apply caching.** To evaluate *cedar*'s ability to leverage caching, we allowed *cedar* to automatically place a cache `Pipe`, which materialized all intermediate samples to disk in the first epoch. We then ran multiple epochs of the torch pipelines across three domains (`CV-torch`, `NLP-torch`, and `ASR`); caching is not applicable to `SSD` as it only used random operators. Table 3 shows the throughput (higher is better) of the cached plan after the first epoch, normalized to the throughput achieved by the plan generated by `cedar-remote`. We also report the *next-best* throughput achieved by enumerating all other cache locations. *cedar* was able to find the optimal location to apply caching.

For `CV-torch`, *cedar* cached the result after Decode and Grayscale, but prior to random Cropping, satisfying randomness requirements. This improved throughput by reducing disk I/O (Decode) and compute (Grayscale). For `NLP-torch`, *cedar* cached the tokenized and truncated sample *prior* to Embedding, avoiding overheads of reading large embeddings. Interestingly, *cedar* did *not* cache `ASR`, determining that re-computation was ideal because transforms significantly increased data volumes. Table 3 confirms this; the next "best" solution cached the output of the entire pipeline, which *reduced* throughput by 18%. *cedar* is effectively able to apply caching while reasoning about its complex interactions with other optimizations.

6.3 In-depth Analysis

Combining optimizations is essential to *cedar*'s performance. To understand *cedar*'s ability to *combine* optimizations, we performed an ablation study by successively enabling local parallelism (i.e., multiple `Drivers`), reordering, offloading (enabling the `distributed` engine), and fusion. All experiments used prefetching.

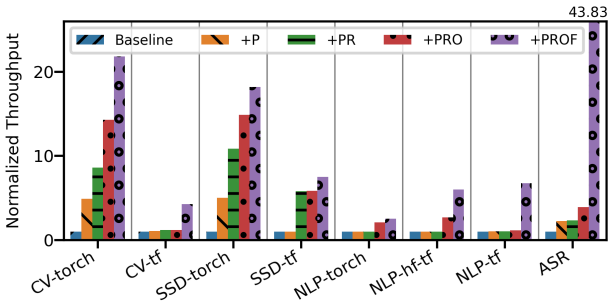

Figure 11: Ablation study showing throughput across pipelines as optimization techniques are successively enabled, normalized to the unoptimized pipeline. P = local parallelism, R = reordering, O = offloading, F = fusion.

Table 4: Throughput overheads with tagging/tracing enabled.

	CV-torch	SSD-torch	NLP-torch	ASR
Throughput Loss	1.45%	2.08%	7.12%	0.71%

Figure 11 shows the throughput of each experiment, normalized to the baseline (i.e., executing the unoptimized `DataSet` within a single `Driver`). No single optimization is a panacea. Instead, a diverse set of optimizations is needed to achieve high performance due to the diverse characteristics across pipelines.

Local parallelism was effective for `CV-torch`, `SSD-torch`, and `ASR` since using multiple `Drivers` bypassed their GIL bottleneck. `CV-tf` and `SSD-tf` largely used multithreaded C++ operators, limiting this benefit. *cedar* did not parallelize the NLP pipelines to avoid serialization overheads. As we explore next, reordering was effective at improving both `CV` and `SSD` pipelines since they used size-changing operators. Meanwhile, `ASR` had more limited reordering opportunities, and the `NLP` pipelines were not able to be reordered. Most pipelines took advantage of offloading across engines, but the overheads incurred by data movement across offloaded pipes limited its effectiveness for some pipelines. By eliminating these overheads with fusion, *cedar* was ultimately able to improve throughput by $2.54 - 43.82\times$ compared to the baseline. The successive improvements with each step showcase *cedar*'s ability to systematically apply and combine optimizations.

Reordering improves throughput by eliminating wasted work on a per-sample basis. While reordering can reduce wasted work based on operator selectivity (see Section 5.1), akin to predicate pushdown in traditional query optimizers, *cedar* further extends this benefit to operators that affect the size of individual samples. The impact of this is shown in Figure 11; reordering increased throughput by up to 5.79× over parallelism alone. Specifically, for the `CV` pipelines, *cedar* reordered operators that reduced the size of each image (Crop, Grayscale) towards the beginning of the pipeline, and size-increasing operators (Float) towards the end. Meanwhile for the `SSD` pipelines, *cedar* moved Distort before Resized Crop, as the Resized Crop increased image sizes. This significantly reduced the necessary compute for each sample while obeying the dependency constraints of each pipeline.

***cedar* introduces minimal overheads.** Finally, we evaluate tracing and `Optimizer` overheads using the PyTorch pipelines of each domain (TF pipelines show similar results). Table 4 reports the throughput overheads introduced by tagging each sample with metadata (to ensure correctness) and periodically tracing samples with statistics (100ms throughout our experiments). While the overheads are slightly larger for high samples/s pipelines (i.e., NLP), these operations introduce minimal overall overheads.

The `Optimizer` can also quickly explore optimizations. As discussed in Section 5.1, the `Optimizer` prunes the search space after each optimization pass, limiting the amount of plans it must evaluate. For the `CV`, `SSD`, `NLP`, and `ASR` PyTorch pipelines, the `Optimizer` considered 251055, 5689, 101954, and 22741 plans, respectively. The `Optimizer` was able to generate a solution in < 6 seconds in each case, insignificant compared to long-running training jobs.

7 RELATED WORK

ML Input Data Frameworks and Optimizations. PyTorch DataLoaders [78] and tf.data [69] are native frameworks for PyTorch and TensorFlow, respectively. DataLoaders offer certain options for performance tuning such as multiprocessing and pinned memory, but requires manual configuration. TorchData [76] is a beta PyTorch data loading library and provides similar primitives to `Pipes`. Its DataLoader2 [74] is an incomplete prototype with an API for distributed processing, but active development has unfortunately stopped [75]. Meanwhile, tf.data can statically fuse and vectorize TF-native pipelines and optimize the CPU and RAM allocation to each operator. Plumber [50] extends tf.data to use a linear program for resource allocation. These frameworks only support local processing, limiting their ability to mitigate data stalls.

DPP [94] and GoldMiner [93] are proprietary distributed services deployed at Meta and Alibaba, respectively. Ray Data [81] is an input data library built on top of Ray [66]. Ray Data distributes processing using Ray's Task and Actor primitives, and optimizes for task overheads by fusing operators via fixed rules. tf.data service [8] offloads processing to distributed workers, but cannot support non-TensorFlow UDFs [84]. FastFlow [87] extends tf.data service to split processing between local and remote workers at a coarse granularity. Pecan [32] is a concurrent work, built on top of tf.data service (and is thus applicable to only TensorFlow pipelines), that also studies transformation ordering, but does not support its concurrent application alongside other optimization passes. These systems, like *cedar*, use an auto-scaling policy to tune worker parallelism.

Various systems extend input data frameworks to address orthogonal concerns. Cachew [31] extends tf.data service to create a service for multi-tenant environments, scaling processing and sharing cached samples between training jobs. Cachew can identify ideal cache locations, but requires users to explicitly insert autocache operators, hindering its compatibility with other optimizations (e.g., reordering). Cachew relies on tf.data's underlying optimizer, which we evaluated in Section 6. PRESTO [41] is a profiler that determines the ideal location to cache, but requires users to manually implement suggestions. CoorDL [65], OneAccess [45], Quiver [51], Tectonic-Shift [96], and SiloD [92] provide distributed caches for data shared across training jobs.

NVIDIA DALI [70] and FusionFlow [47] use GPUs for input data processing. Revamper [54], SHADE [46], and iCACHE [20] introduce optimizations that modify input data semantics (e.g., upsampling) to improve model convergence and input data efficiency. RecD [95] deduplicates recommendation datasets for input data processing efficiency. *cedar*'s extensibility allows it to easily adopt these engines and techniques alongside its current optimizations.

As discussed in Section 3.1, *cedar* addresses the key requirements that are not met by current input data systems – systematic, context-aware, and general optimizations. *cedar* provides an easy-to-use programming interface, supporting general ML frameworks and pipelines, that permits users to express a simple set of constraints that unlock a rich set of reordering and caching optimizations. Meanwhile, its `Optimizer` systematically applies a complex set of optimizations to improve performance, and its `Scaler` leverages these benefits to efficiently meet training demand.

Traditional Processing Frameworks. Naiad [68], Spark [90], DryadLINQ [89], and many other processing frameworks [2, 3, 13, 24, 40, 60] allow users to chain together higher-order transformations and model computation as data flows. Other data processing frameworks, such as Apache Beam [2, 28], Apache Wayang [11], and RHEEM [49], leverage an optimizer and decouple dataflow graphs from underlying execution engines, similar to *cedar*'s `Execution` interface. *cedar* is specialized for ML input data pipelines.

cedar applies similar rule- and cost-based optimization passes to traditional query optimizers [7, 29, 30], while also considering unique properties to ML input data pipelines such as relaxed order dependencies and randomness. Lara [52] is a domain-specific language that optimizes matrix-heavy preprocessing for traditional "shallow" ML models (e.g., regressions). Hueske *et al.* [38] explore preserving UDF semantics under reordering by statically analyzing PACT [10] programs. *cedar* extends the impact of these traditional data processing techniques to ML input data systems.

8 CONCLUSION

We presented *cedar*, a unified framework to define, optimize, and execute ML input data pipelines. *cedar*'s `Feature` API allows users to define input data pipelines using modular `Pipes` and to express lightweight hints that allow *cedar* to reason about operator semantics. *cedar* automatically applies systematic, context-aware, and general optimizations to improve performance, and it orchestrates pipeline execution to efficiently meet training throughput demands. *cedar* outperforms tf.data, tf.data service, FastFlow, Plumber, Ray Data, and PyTorch DataLoader by up to 1.87× to 10.65× across diverse pipelines. *cedar* provides extensible and general programming, optimizer, and execution interfaces that allow it to enable and evolve alongside future ML input data systems research.

ACKNOWLEDGMENTS

We gratefully acknowledge Johann Hauswald, Andrew Woen, and our anonymous reviewers whose feedback has greatly helped improve this paper. This research was partly 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 and a Meta PhD Fellowship.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] 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. *Proceedings of the VLDB Endowment* 8 (2015), 1792–1803.

[3] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. 2014. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal* 23, 6 (dec 2014), 939–964. https://doi.org/10.1007/s00778-014-0357-y

[4] Gene M. Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference* (Atlantic City, New Jersey) *(AFIPS '67 (Spring))*. Association for Computing Machinery, New York, NY, USA, 483–485. https://doi.org/10.1145/1465482.1465560

[5] Rosana Ardila, Megan Branson, Kelly Davis, Michael Henretty, Michael Kohler, Josh Meyer, Reuben Morais, Lindsay Saunders, Francis M. Tyers, and Gregor Weber. 2020. Common Voice: A Massively-Multilingual Speech Corpus. arXiv:1912.06670 [cs.CL]

[6] 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

[7] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1383–1394. https://doi.org/10.1145/2723372.2742797

[8] Andrew Audibert, Yang Chen, Dan Graur, Ana Klimovic, Jiří Šimša, and Chandramohan A. Thekkath. 2023. Tf.Data Service: A Case for Disaggregating ML Input Data Processing. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '23)*. Association for Computing Machinery, New York, NY, USA, 358–375. https://doi.org/10.1145/3620678.3624666

[9] AWS. 2024. AWS Trainium. https://aws.amazon.com/machine-learning/trainium/. Accessed 2024-10-21.

[10] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. 2010. Nephele/PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 119–130. https://doi.org/10.1145/1807128.1807148

[11] Kaustubh Beedkar, Bertty Contreras-Rojas, Haralampos Gavriilidis, Zoi Kaoudi, Volker Markl, Rodrigo Pardo-Meza, and Jorge-Arnulfo Quiané-Ruiz. 2023. Apache Wayang: A Unified Data Analytics Framework. *SIGMOD Rec.* 52, 3 (nov 2023), 30–35. https://doi.org/10.1145/3631504.3631510

[12] Benjamin Berg. 2023. A Principled Approach to Parallel Job Scheduling. (1 2023). https://doi.org/10.1184/R1/21817980.v1

[13] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. 2011. Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE '11)*. IEEE Computer Society, USA, 1151–1162. https://doi.org/10.1109/ICDE.2011.5767921

[14] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs.* http://github.com/google/jax Accessed 2024-10-21.

[15] G. Bradski. 2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools* (2000).

[16] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]

[17] P Carbone, S Ewen, S Haridi, A Katsifodimos, V Markl, and K Tzoumas. 2015. Apache FlinkTM: Stream and batch processing in a single engine. *Bull. IEEE Comput. Soc. Tech. Comm. Data Eng* 36, 4 (2015).

[18] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A Simple Framework for Contrastive Learning of Visual Representations. *arXiv preprint arXiv:2002.05709* (2020).

[19] Ting Chen, Simon Kornblith, Kevin Swersky, Mohammad Norouzi, and Geoffrey Hinton. 2020. Big Self-Supervised Models are Strong Semi-Supervised Learners. *arXiv preprint arXiv:2006.10029* (2020).

[20] Weijian Chen, Shuibing He, Yaowen Xu, Xuechen Zhang, Siling Yang, Shuang Hu, Xian-He Sun, and Gang Chen. 2023. iCache: An Importance-Sampling-Informed Cache for Accelerating I/O-Bound DNN Model Training. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 220–232. https://doi.org/10.1109/HPCA56546.2023.10070964

[21] Papers With Code. 2024. Trends. https://paperswithcode.com/trends. Accessed 2024-10-21.

[22] 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_files/paper/2020/file/d85b63ef0ccb114d0a3bb7b7d808028f-Paper.pdf

[23] 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

[24] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (jan 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[25] 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

[26] Steven Y. Feng, Varun Gangal, Jason Wei, Sarath Chandar, Soroush Vosoughi, Teruko Mitamura, and Eduard Hovy. 2021. A Survey of Data Augmentation Approaches for NLP. arXiv:2105.03075 [cs.CL]

[27] Apache Software Foundation. 2024. Apache Arrow. https://arrow.apache.org. Accessed 2024-10-21.

[28] Apache Software Foundation. 2024. Apache Beam. https://beam.apache.org. Accessed 2024-10-21.

[29] Goetz Graefe. 1995. The cascades framework for query optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29.

[30] G. Graefe and W.J. McKenna. 1993. The Volcano optimizer generator: extensibility and efficient search. In *Proceedings of IEEE 9th International Conference on Data Engineering*. 209–218. https://doi.org/10.1109/ICDE.1993.344061

[31] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A. Thekkath, and Ana Klimovic. 2022. Cachew: Machine Learning Input Data Processing as a Service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 689–706. https://www.usenix.org/conference/atc22/presentation/graur

[32] Dan Graur, Oto Mraz, Muyu Li, Sepehr Pourghannad, Chandramohan A. Thekkath, and Ana Klimovic. 2024. Pecan: Cost-Efficient ML Data Preprocessing with Automatic Transformation Ordering and Hybrid Placement. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 649–665. https://www.usenix.org/conference/atc24/presentation/graur

[33] Alex Graves. 2012. Sequence Transduction with Recurrent Neural Networks. arXiv:1211.3711 [cs.NE] https://arxiv.org/abs/1211.3711

[34] Song Han, Huizi Mao, and William J Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *International Conference on Learning Representations (ICLR)* (2016).

[35] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning Both Weights and Connections for Efficient Neural Networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1* (Montreal, Canada) *(NIPS'15)*. MIT Press, Cambridge, MA, USA, 1135–1143.

[36] Mark D. Hill and Michael R. Marty. 2008. Amdahl's Law in the Multicore Era. *Computer* 41, 7 (2008), 33–38. https://doi.org/10.1109/MC.2008.209

[37] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (nov 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[38] Fabian Hueske, Mathias Peters, Matthias J. Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. 2012. Opening the Black Boxes in Data Flow Optimization. *Proc. VLDB Endow.* 5, 11 (jul 2012), 1256–1267. https://doi.org/10.14778/2350229.2350244

[39] HuggingFace. 2024. Tokenizer Summary. https://huggingface.co/docs/transformers/tokenizer_summary. Accessed 2024-10-21.

[40] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (Lisbon, Portugal) *(EuroSys '07)*. Association for Computing Machinery, New York, NY, USA, 59–72. https://doi.org/10.1145/1272996.1273005

[41] Alexander Isenko, Ruben Mayer, Jeffrey Jedele, and Hans-Arno Jacobsen. 2022. Where Is My Training Bottleneck? Hidden Trade-Offs in Deep Learning Preprocessing Pipelines. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1825–1839. https://doi.org/10.1145/3514221.3517848

[42] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. 2024. MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 745–760. https://www.usenix.org/conference/nsdi24/presentation/jiang-ziheng

[43] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 82, 14 pages. https://doi.org/10.1145/3579371.3589350

[44] D. Juedes, F. Drews, L. Welch, and D. Fleeman. 2004. Heuristic resource allocation algorithms for maximizing allowable workload in dynamic, distributed real-time systems. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* 117–. https://doi.org/10.1109/IPDPS.2004.1303072

[45] Aarati Kakaraparthy, Abhay Venkatesh, Amar Phanishayee, and Shivaram Venkataraman. 2019. The Case for Unifying Data Loading in Machine Learning Clusters. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/hotcloud19/presentation/kakaraparthy

[46] Redwan Ibne Seraj Khan, Ahmad Hossein Yazdani, Yuqi Fu, Arnab K. Paul, Bo Ji, Xun Jian, Yue Cheng, and Ali R. Butt. 2023. SHADE: Enable Fundamental Cacheability for Distributed Deep Learning Training. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 135–152. https://www.usenix.org/conference/fast23/presentation/khan

[47] Taeyoon Kim, ChanHo Park, Mansur Mukimbekov, Heelim Hong, Minseok Kim, Ze Jin, Changdae Kim, Ji-Yong Shin, and Myeongjae Jeon. 2024. FusionFlow: Accelerating Data Preprocessing for Machine Learning with CPU-GPU Cooperation. *Proc. VLDB Endow.* 17, 4 (mar 2024), 863–876. https://doi.org/10.14778/3636218.3636238

[48] Jan Kossmann, Thorsten Papenbrock, and Felix Naumann. 2021. Data Dependencies for Query Optimization: A Survey. *The VLDB Journal* 31, 1 (jun 2021), 1–22. https://doi.org/10.1007/s00778-021-00676-3

[49] Sebastian Kruse, Zoi Kaoudi, Bertty Contreras-Rojas, Sanjay Chawla, Felix Naumann, and Jorge-Arnulfo Quiané-Ruiz. 2020. RHEEMix in the data jungle: a cost-based optimizer for cross-platform systems. *The VLDB Journal* 29, 6 (may 2020), 1287–1310. https://doi.org/10.1007/s00778-020-00612-x

[50] Michael Kuchnik, Ana Klimovic, Jiri Simsa, Virginia Smith, and George Amvrosiadis. 2022. Plumber: Diagnosing and removing performance bottlenecks in machine learning data pipelines. *Proceedings of Machine Learning and Systems* 4 (2022), 33–51.

[51] 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

[52] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. An Intermediate Representation for Optimizing Machine Learning Pipelines. *Proc. VLDB Endow.* 12, 11 (jul 2019), 1553–1567. https://doi.org/10.14778/3342263.3342633

[53] Frederic Lardinois. 2022. 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/. Accessed 2024-10-21.

[54] 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

[55] Qian Li, Bin Li, Pietro Mercati, Ramesh Illikkal, Charlie Tai, Michael Kishinevsky, and Christos Kozyrakis. 2021. RAMBO: Resource Allocation for Microservices Using Bayesian Optimization. *IEEE Computer Architecture Letters* 20, 1 (2021), 46–49. https://doi.org/10.1109/LCA.2021.3066142

[56] 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] https://arxiv.org/abs/1405.0312

[57] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. 2016. *SSD: Single Shot MultiBox Detector.* Springer International Publishing, 21–37. https://doi.org/10.1007/978-3-319-46448-0_2

[58] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim Hazelwood, Andrew Hock, Xinyuan Huang, Atsushi Ike, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Tsuguchika Tabaru, Carole-Jean Wu, Lingjie Xu, Masafumi Yamazaki, Cliff Young, and Matei Zaharia. 2019. MLPerf Training Benchmark. arXiv:1910.01500 [cs.LG]

[59] Brian McFee, Colin Raffel, Dawen Liang, Daniel P Ellis, Matt McVicar, Eric Battenberg, and Oriol Nieto. 2015. librosa: Audio and Music Signal Analysis in Python. In *Proceedings of the 14th Python in Science Conference*, Vol. 8. 18–25.

[60] Erik Meijer, Brian Beckman, and Gavin Bierman. 2006. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (Chicago, IL, USA) *(SIGMOD '06)*. Association for Computing Machinery, New York, NY, USA, 706. https://doi.org/10.1145/1142473.1142552

[61] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer Sentinel Mixture Models. arXiv:1609.07843 [cs.CL]

[62] Meta. 2022. Introducing the AI Research SuperCluster. https://ai.facebook.com/blog/ai-rsc/. Accessed 2024-10-21.

[63] Meta. 2024. Building Meta's GenAI Infrastructure. https://engineering.fb.com/2024/03/12/data-center-engineering/building-metas-genai-infrastructure/. Accessed 2024-10-21.

[64] MindSpore. 2024. MindSpore. https://github.com/mindspore-ai/mindspore. Accessed 2024-10-21.

[65] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. 2021. Analyzing and Mitigating Data Stalls in DNN Training. *Proc. VLDB Endow.* 14, 5 (jan 2021), 771–784. https://doi.org/10.14778/3446095.3446100

[66] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 561–577.

[67] 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 *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 993–1011. https://doi.org/10.1145/3470496.3533727

[68] 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

[69] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. 2021. Tf.Data: A Machine Learning Data Processing Framework. *Proc. VLDB Endow.* 14, 12 (jul 2021), 2945–2958. https://doi.org/10.14778/3476311.3476374

[70] NVIDIA. 2024. NVIDIA DALI. https://docs.nvidia.com/deeplearning/dali/user-guide/docs/index.html. Accessed 2024-10-21.

[71] 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

[72] Daniel S. Park, William Chan, Yu Zhang, Chung-Cheng Chiu, Barret Zoph, Ekin D. Cubuk, and Quoc V. Le. 2019. SpecAugment: A Simple Data Augmentation Method for Automatic Speech Recognition. In *Proc. Interspeech 2019*. 2613–2617. https://doi.org/10.21437/Interspeech.2019-2680

[73] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach 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*. Curran Associates Inc., Red Hook, NY, USA.

[74] PyTorch. 2024. DataLoader2. https://github.com/pytorch/data/blob/a5b4720dece60565788ac4c9a85e01719188b28e/torchdata/dataloader2/README.md. Accessed 2024-10-21.

[75] PyTorch. 2024. Future of torchdata and dataloading. https://github.com/pytorch/data/issues/1196. Accessed 2024-10-21.

[76] PyTorch. 2024. TorchData. https://pytorch.org/data/beta/index.html. Accessed 2024-10-21.

[77] PyTorch. 2024. Torchtext. https://torchtext.readthedocs.io/en/latest/. Accessed 2024-10-21.

[78] PyTorch. 2024. torch.utils.data. https://pytorch.org/docs/stable/data.html. Accessed 2024-10-21.

[79] PyTorch. 2024. Torchvision. https://pytorch.org/vision/stable/index.html. Accessed 2024-10-21.

[80] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).

[81] Ray. 2024. Ray Data: Scalable Datasets for ML. https://docs.ray.io/en/latest/data/data.html. Accessed 2024-10-21.

[82] Dave Salvator. 2022. NVIDIA Hopper, Ampere GPUs Sweep Benchmarks in AI Training. https://blogs.nvidia.com/blog/2022/11/09/mlperf-ai-training-hpc-hopper/. Accessed 2024-10-21.

[83] 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 '22)*. Association for Computing Machinery, New York, NY, USA, 344–358. https://doi.org/10.1145/3503222.3507777

[84] TensorFlow. 2023. tf.data.experimental.service. https://www.tensorflow.org/versions/r2.14/api_docs/python/tf/data/experimental/service#limitations. Accessed 2024-10-21.

[85] TensorFlow. 2024. TensorFlow Graph Optimization. https://www.tensorflow.org/guide/graph_optimization. Accessed 2024-10-21.

[86] TensorFlow. 2024. TensorFlow Text. https://www.tensorflow.org/text. Accessed 2024-10-21.

[87] Taegeon Um, Byungsoo Oh, Byeongchan Seo, Minhyeok Kweun, Goeun Kim, and Woo-Yeon Lee. 2023. FastFlow: Accelerating Deep Learning Model Training with Smart Offloading of Input Data Pipeline. *Proc. VLDB Endow.* 16, 5 (jan 2023), 1086–1099. https://doi.org/10.14778/3579075.3579083

[88] Erlin Yao, Yungang Bao, Guangming Tan, and Mingyu Chen. 2009. Extending Amdahl's law in the multicore era. *SIGMETRICS Perform. Eval. Rev.* 37, 2 (oct 2009), 24–26. https://doi.org/10.1145/1639562.1639571

[89] 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.

[90] 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

[91] Seyed Majid Zahedi, Qiuyun Llull, and Benjamin C. Lee. 2018. Amdahl's Law in the Datacenter Era: A Market for Fair Processor Allocation. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 1–14. https://doi.org/10.1109/HPCA.2018.00011

[92] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Mingxia Li, Fan Yang, Qianxi Zhang, Binyang Li, Yuqing Yang, Lili Qiu, Lintao Zhang, and Lidong Zhou. 2023. SiloD: A Co-Design of Caching and Scheduling for Deep Learning Clusters. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 883–898. https://doi.org/10.1145/3552326.3567499

[93] Hanyu Zhao, Zhi Yang, Yu Cheng, Chao Tian, Shiru Ren, Wencong Xiao, Man Yuan, Langshi Chen, Kaibo Liu, Yang Zhang, Yong Li, and Wei Lin. 2023. GoldMiner: Elastic Scaling of Training Data Pre-Processing Pipelines for Deep Learning. *Proc. ACM Manag. Data* 1, 2, Article 193 (jun 2023), 25 pages. https://doi.org/10.1145/3589773

[94] 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. 2022. 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* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 1042–1057. https://doi.org/10.1145/3470496.3533044

[95] Mark Zhao, Dhruv Choudhary, Devashish Tyagi, Ajay Somani, Max Kaplan, Sung-Han Lin, Sarunya Pumma, Jongsoo Park, Aarti Basant, Niket Agarwal, Carole-Jean Wu, and Christos Kozyrakis. 2023. RecD: Deduplication for End-to-End Deep Learning Recommendation Model Training Infrastructure. In *Proceedings of Machine Learning and Systems*, D. Song, M. Carbin, and T. Chen (Eds.), Vol. 5. Curan, 754–767. https://proceedings.mlsys.org/paper_files/paper/2023/file/f9b15fec25182f2d70af68a39546d60e-Paper-mlsys2023.pdf

[96] Mark Zhao, 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, Carole-Jean Wu, and Christos Kozyrakis. 2023. Tectonic-Shift: A Composite Storage Fabric for Large-Scale ML Training. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 433–449. https://www.usenix.org/conference/atc23/presentation/zhao