MIXTERA: A Data Plane for Foundation Model Training

Maximilian Böther
ETH Zurich
Switzerland
mboether@ethz.ch

Xiaozhe Yao
ETH Zurich
Switzerland
xiayao@ethz.ch

Tolga Kerimoglu
ETH Zurich
Switzerland
tkerimoglu@student.ethz.ch

Ana Klimovic
ETH Zurich
Switzerland
aklimovic@ethz.ch

ABSTRACT

State-of-the-art large language and vision models are trained over trillions of tokens that are aggregated from a large variety of sources. As training data collections grow, manually managing the samples becomes time-consuming, tedious, and prone to errors. Yet recent research shows that the data mixture and the order in which samples are visited during training can significantly influence model accuracy. We build and present MIXTERA, a data plane for foundation model training that enables users to declaratively express which data samples should be used in which proportion and in which order during training. MIXTERA is a centralized, read-only layer that is deployed on top of existing training data collections and can be declaratively queried. It operates independently of the filesystem structure and supports mixtures across arbitrary properties (e.g., language, source dataset) as well as dynamic adjustment of the mixture based on model feedback. We experimentally evaluate MIXTERA and show that our implementation does not bottleneck training and scales to 256 GH200 superchips. We demonstrate how MIXTERA supports recent advancements in mixing strategies by implementing the proposed Adaptive Data Optimization (ADO) algorithm in the system and evaluating its performance impact. We also explore the role of mixtures for vision-language models.

PVLDB Reference Format:
Maximilian Böther, Xiaozhe Yao, Tolga Kerimoglu, and Ana Klimovic. MIXTERA: A Data Plane for Foundation Model Training. PVLDB, 1(1): 1 - 1, 1. doi:1

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

1 INTRODUCTION

Large language and vision models (often called foundation models) have become omnipresent in our daily life. They show enormous capabilities in a diverse set of tasks [8, 9, 38, 46, 52], such as assistance with writing and coding, video understanding, and even agentic interaction with the world. The training of such language and vision models (LLMs/VLMs) presents new challenges for managing training data due to the ever-growing sizes of models and datasets. To achieve high accuracy, state-of-the-art models train over trillions of tokens from aggregated data collections such as RedPajama [68], Dolma [60], or FineWeb [48]. For example, Meta's Llama 3.3 70B model is trained on a corpus of 15 trillion tokens [38, 39]. These collections are built based on data from various sources, such as Wikipedia or Common Crawl dumps.

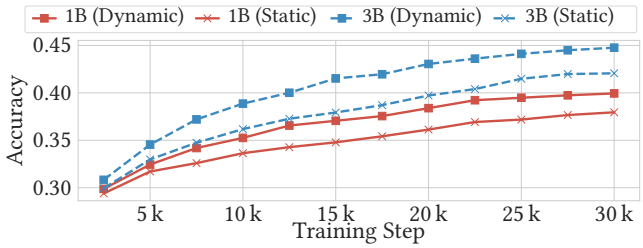
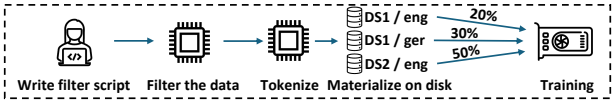


Figure 1: Dynamically adjusting the mixture using the ADO algorithm improves pre-training performance on HellaSwag over the default static mixture across model scales.

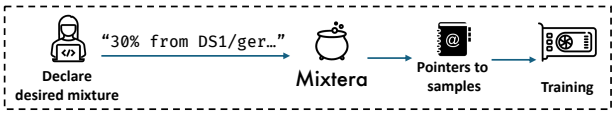
Training data is typically stored on distributed filesystems in GPU clusters or data lakes in the cloud, and managed manually and ad hoc by ML engineers (Figure 2a). Selecting the right proportions out of this data with particular characteristics (e.g., language, topic, source) for training is critical for model performance [12, 69, 72]. Individual users write ad hoc scripts to process the training data, filter relevant subsets with the properties of interest, often pre-tokenize it, and then mix it for their use case, e.g., they might want to train on 50 % data from Wikipedia and 50 % from movie subtitles. This reflects our experience working with ML researchers as part of the Swiss AI initiative which aims to develop open-source LLMs [16]. We confirmed through discussions with industry teams that such setups are common. This can get complex as training data may need to be mixed based on a variety of characteristics. For example, in addition to satisfying source data proportions (Wikipedia vs. movie subtitles in the previous example), we may also want the training data to be 80 % in English and 20 % in German. Recent works show that the data mixture should also be adjusted during training. The SMOLLM2 model is trained with four stages of data mixtures [4]. Algorithms such as Adaptive Data Optimization (ADO) [26], AIOLI [11], PIKE [31], and SKILL-IT [12] even propose to adjust the mixture dynamically based on the model behavior (e.g., loss per domain) during training. Figure 1 shows that using ADO can increase the performance of LLMs over a static mixture across different scales on the downstream HellaSwag benchmark [75].

Today, there is no open-source system that automatically manages vast amounts of training data based on fine-grained characteristics. Implementing data filtering and subsequent mixing requires users to manually keep track of metadata. At least in the open-source world, this is typically done as part of the directory structure of the filesystem, e.g., there is one subdirectory per source dataset, and then we sample from each directory (Figure 2a). This approach is limited because each data sample has multiple properties that can

Proceedings of the VLDB Endowment, Vol. 1, No. 1 ISSN 2150-8097.
doi:1



(a) Without Mixtera: More developer, CPU, and disk resources needed.



(b) With Mixtera: Declarative mixture specification and low overhead.

Figure 2: Mixtera needs less offline processing and scripting.

be used to determine whether it should be used for training. Filesystems fundamentally do not offer the right interface for managing training data and mixing, as they do not provide declarative query interfaces or a native way to track which model was trained on what data. Running a data processing job to fully materialize the mixed training set for each training run has lots of overhead and leads to data duplication. A streaming-oriented approach that selects, tokenizes, and mixes data on the fly provides much more flexibility and eases experimentation. Utilizing a full-fledged DBMS for tracking data properties would burden ML engineers with complexities such as database administration, schema design, and performance tuning. We need a lightweight data plane that enables users to declaratively query and mix data based on arbitrary properties, independent of the filesystem structure, and to adjust this mixture dynamically.

We present Mixtera, a data plane that can be deployed on top of existing LLM/VLM training data collections. It is a centralized, read-only layer and can be declaratively queried from training clients. As shown in Figure 2b, it reduces the amount of materialization and data scripting necessary for training jobs. While existing data loaders as outlined in Table 1 are limited by filesystem constraints, Mixtera supports arbitrary properties independent of the filesystem, and makes it easy for model engineers to experiment with different filter criteria, fixed learning curricula, and fully dynamic mixing algorithms that learn the mixture during training. Mixtera follows a client-server model. For a training job, the server, which indexes and manages the samples, first statically filters out the relevant samples, and then continuously distributes *chunks* to the clients. Chunks are fixed-size collections of *pointers* to samples in files, and adhere to the current mixture. The clients then fetch the relevant samples and return the relevant data to the training loop. We design and implement Mixtera tailored to the needs for foundation model training, and make the following key contributions:

(1) Mixtera enables users to declaratively specify mixtures across properties independent of the filesystem structure and training frameworks by indexing all samples and their properties.
(2) Mixtera enables dynamically changing the data mixture and the properties determining the mixture during training. It achieves this by generating and streaming chunks, i.e., fixed-size lists of pointers to samples following the current mixture.
(3) We show Mixtera does not bottleneck training and is versatile, enabling model accuracy improvements for text-only and text-vision models. As an example, we demonstrate how to implement the ADO dynamic data mixing algorithm in Mixtera and its positive impact on model accuracy.

2 BACKGROUND

Foundation models are large-scale deep learning models suitable for a variety of tasks [8, 14]. We focus on text-generation models, i.e., autoregressive large language models (LLMs) and multimodal vision-language models (VLMs). As of 2025, most such models are based on the Transformer architecture [66]. They are trained on huge corpora of training data in a self-supervised manner to maximize the likelihood of predicting the tokens of a training sequence.

Training phases. Training is structured into *pre-training* and *post-training* phases. In pre-training, we train a randomly initialized model on a general purpose data corpus (Section 2.1) to derive a *base model*. In post-training, common steps include *supervised finetuning* (SFT) and *alignment*. The data used in SFT is human-selected for a specific task. In this paper, we focus on the pre-training use case.

Distributed training. Training foundation models requires distributing computation across multiple GPUs. Training frameworks typically employ 3D parallelism [6, 21], consisting of pipeline parallelism (PP), i.e., partioning the model layers between devices [22, 44, 45], tensor parallelism (TP), i.e., splitting individual tensor operations within layers across devices [55, 57], and data parallelism (DP), i.e., replication of the model across device groups. PP and TP together are referred to as *model parallelism*. Nodes within the same DP group process identical inputs, while nodes across DP groups receive different data. As an extension to DP, fully-sharded data parallelism (FSDP) shards model parameters, gradients, and optimizer states across data-parallel workers [78].

2.1 Training Data and Data Mixing

The data used for pre-training stems from data collections that include data from various sources, such as Wikipedia, Common Crawl dumps, or arXiv papers. Public examples of such collections include RedPajama [68], Dolma [60], and FineWeb [48]. Besides the

Table 1: Feature comparison of Mixtera and other open-source data loaders.

	Mixtera	HF Datasets	WebDatasets	Mosaic Streaming
File formats	jsonl(.zst), parquet, webdataset	jsonl(.zst), parquet, webdataset, csv	webdataset	Mosaic Data Shard, jsonl, csv
Static filtering	declarative	using map UDFs	using map UDFs	using map UDFs
Static mixtures	on all properties	on filesystem dirs.	on filesystem dirs.	on filesystem dirs.
Dynamic mixtures	on all properties	no	no	no
Native 3D parallelism	yes	yes, manual rank handling	data parallel only	yes, for specific rank order
Checkpointing	yes	using TorchData	by replaying, not natively	yes

aggregation of data from different sources, data engineers typically clean the data, which typically involves deduplicating, filtering (e.g., the removal of personal identifiable information - PII), and applying classifiers to the data samples (e.g., to obtain a toxicity score for each sample). Longpre et al. [36] and Penedo et al. [48] provide a great overview of the processing steps and their impact.

Data properties and mixtures. Each sample in the resulting collection has properties, such as its source (e.g., Wikipedia) or its language (e.g., English). Beyond the filtering operations, ML engineers need to define a *data mixture*. A mixture describes how the data is mixed based on their characteristics, i.e., we might train on 50 % data from Common Crawl and 50 % from movie subtitles. The data can be combined based on multiple characteristics simultaneously. For instance, besides Common Crawl and movie subtitles, we might also use 80 % French and 20 % Italian data. The granularity on which the mixture is ensured depends on each training setup and there is no common agreement. For example, a mixture could be ensured within a batch, or across a window of several batches.

Mixing algorithms. Selecting the best mixture is critical for model performance [12, 56, 69, 72]. We differentiate *static mixtures*, i.e., mixtures that remain constant over the entire training job, and *dynamic mixtures*, i.e., mixtures that change during the training job. In the last years, several algorithms for finding the best static mixture or how to adjust the mixture dynamically during training have been proposed. Algorithms such as DoReMi [69] or the data mixing laws [72] find the a static mixture via small proxy models.

Curriculum learning is an example of a pre-defined dynamic mixture. Xu et al. [70] order samples from *easy* to *hard* to improve alignment. The SmolLM2 model was trained on 4 stages of mixtures [4]. Multilingual models are often first trained on English data, and then data from other languages is included [53, 71].

Beyond such pre-defined schedules, there is also work on adapting the data mixture to the model training dynamics, e.g., by increasing the weight of data domains which have high loss. Albalak et al. [3] use a multi-armed bandit strategy to adjust the mixture. Skill-it orders "skills" based on model feedback [12]. Aioli builds upon Skill-it and provides a unified framework for estimating the best mixture during training [11]. PiKE relies on gradient interactions [31]. In this paper, we use Adaptive Data Optimization (ADO) [26] as an example of a dynamic mixing algorithms.

2.1.1 Adaptive Data Optimization. Adaptive Data Optimization (ADO) is a dynamic mixing algorithm that adjusts the data mixture during training based on the model's learning progress on each domain [26]. The key idea is to prioritize domains where the model shows rapid improvement while considering how much each domain contributes to its own progress. ADO uses neural scaling laws to model how the loss L_k of each domain k decreases with the number of training samples n. To this end, it fits a power law $\hat{L}_k(n) = \varepsilon_k + \beta_k n^{-\alpha_k}$ *for each domain*. Here, ε_k represents the irreducible loss of the domain, β_k is a scaling factor, and α_k determines how quickly the loss decreases. The parameters are re-fitted during training. The algorithm combines two components to determine the mixture weights. First, it estimates the learning speed for each domain using the derivative of the scaling law. Second, ADO maintains a credit assignment score $\lambda_k(t)$ that indicates how much each domain contributes to its own progress, based on its recent sampling frequency. These components are combined with a prior (initial) distribution μ_k to compute an intermediate preference distribution $\rho_k(t)$. To ensure stability, the final distribution $\pi_k(t)$ is then computed as a weighted average between $\rho_k(t)$ and $\pi_k(t)$'s temporal average. Additionally, ADO enforces a minimum sampling probability for each domain. For more details, we refer to Jiang et al. [26].

3 CURRENT CHALLENGES

We identify three challenges in the status quo of training data management with current open-source infrastructure.

Challenge 1: High engineering effort for data preparation. The current approach to preparing training data sets involves many manual offline steps with general-purpose data processing and scripting frameworks (Figure 2a). For the offline cleaning step (Section 2.1), ML engineers typically leverage data processing frameworks like Spark [74], Beam [1], Data-Juicer [10], or datatrove [49].

The subsequent data mixing can happen offline or online. If data is mixed offline, engineers write ad hoc scripts which create a new mixed copy of the cleaned dataset for each training run. This makes it difficult to get a quick sense of how a data mixture will impact model training when exploring different mixing policies. Some existing data loaders support online data mixing. However, they only implement it across directories, i.e., they assign a weight to each directory and sample data according to the weights. Hence, the directories need to reflect the property we want to mix on, e.g., one directory per language or data source. This inflexible approach neither supports switching the property we mix on nor supports specifying hierarchical mixtures across arbitrary properties, e.g., specify a proportion between Wikipedia and movie subtitles, *and* between English and German (c.f. Table 1).

Challenge 2: Inefficient data management on filesystems. Training data is typically stored and managed as files without a proper data management system, which can lead to *storage overhead, performance bottlenecks, and consistency issues*. Both the source data and mixed training set are typically stored in formats such as `jsonl` or `parquet` files in a shared, distributed filesystem. Existing data loaders just wrap around the filesystem and are therefore limited by its constraints (Table 1). Filesystems are commonly used because current database management systems are not natively built for foundation model training data [67]. Using a proper DBMS for such data would require ML developers to define table schemas and to orchestrate dataflow from the DBMS to the model training, in addition to administration overhead to operate the DBMS itself. This situation is problematic, as (i) both the metadata and actual payloads are commonly duplicated across training jobs, (ii) filesystems do not provide an efficient interfaces to query data, (iii) there is no native way of tracking which model was trained on which data if it is accessed via general-purpose filesystem calls. Some frameworks like Megatron [45, 57] even require pre-tokenized data, which leads to duplication of the even bigger tokenized data files.

Challenge 3: Rapidly evolving research. How to train the best model on a given dataset is an active area of research, with many new techniques such as dynamic mixing emerging. Offline preparation of the mixture or online mixing based on fixed directory weights does not support dynamic mixture at all. Even for

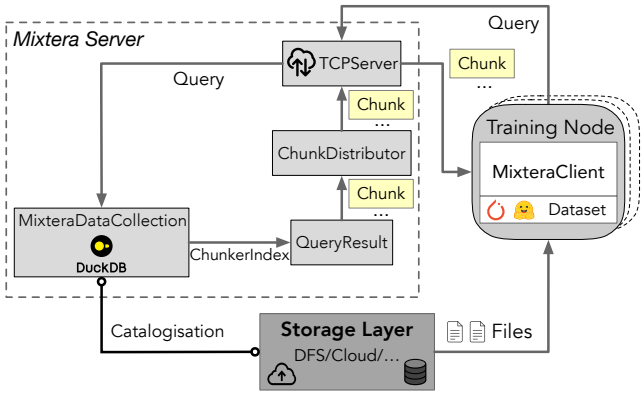


Figure 3: Mixtera system architecture.

```python
client = MixteraClient("127.0.0.1", 8080)
job_id = "test_job"
query = Query.for_job(job_id).select(("license","==","CC"))
mixture = StaticMixture(
    { MixtureKey({"language": ["JavaScript"]}): 0.7,
     MixtureKey({"language": ["HTML"]}): 0.3 },
    chunk_size=1024)
qea = QueryExecutionArgs(mixture=mixture, num_workers=4,
                         dp_groups=1, nodes_per_group=1)
rsa = ResultStreamingArgs(node_id=0, dp_group_id=0, job_id=job_id)
ds = MixteraTorchDataset(client, query, rea, rsa)
dl = torch.utils.data.DataLoader(ds, batch_size=1024, num_workers=4)

for batch in dl:
    print(batch)
```

Figure 4: An example query using Mixtera.

researchers who are familiar with the latest mixing techniques, implementing modern mixing algorithms in a training pipeline is a painful, tedious, and error-prone task. The codebases for mixing algorithms are often tailored directly to the training framework, as well as the data collection and properties used in the respective papers. This hinders the adoption of new methods and makes researching, reproducing, and comparing different strategies difficult.

4 MIXTERA'S DESIGN

We propose to address these challenges for training data management by building Mixtera, a foundation model training data plane. We derive the following design goals for such a system.

Goal G1: The system should implement a *centralized* data layer that users can conveniently and declaratively query to mix data across arbitrary properties, independent of the filesystem structure.

Goal G2: The system needs to be *lightweight*, i.e., not require many components to set up and be easily integratable into existing training setups.

Goal G3: While being user-friendly and flexible, the system needs to ensure *high-throughput* and *determinism* (reproducibility).

Goal G4: The system must support *adjusting the mixture dynamically* during the training.

4.1 System Overview and Design

Inspired by the Lakehouse architecture [5], we design Mixtera as a read-only layer that can be deployed on top of existing training data collections, which are typically stored in a distributed filesystem. Figure 3 shows the architecture of the system. Mixtera manages a centralized database of metadata (i.e., properties such as language or source dataset) of all training data samples (G1). A sample can be a piece of text (for LLM training) or a text-image pair (for VLM training). Mixtera assigns every sample a unique ID and properties instead of treating the training data as a blob of homogeneous, contiguous data. It allows users to declaratively query the relevant samples for a training. To be lightweight (G2), Mixtera does not reorganize or modify the data files on disk. For model training, it provides a standard iterator that can be used, e.g., in conjunction with a `torch.DataLoader`. Mixtera is agnostic to the model training framework, supports training interruptions using checkpoints, and ensures determinism (G3) through careful shuffling, i.e., for identical queries, Mixtera always provides data in an identical order, which is important for reproducibility as well as for debugging issues like loss spikes [15, 27, 50, 64, 79]. It supports adjusting the mixture during training (G4) by transferring chunks (lists of pointers to samples) whose mixture can change over time.

Query interface. In Figure 4, we show an example of a query that statically selects only Creative Commons data, and then mixes HTML and JavaScript data in a 70:30 ratio during training. Mixtera's implementation takes care of executing the query and obtaining the samples without needing to worry about correctness, even in distributed training. The user only needs to provide the ID of the node and its data parallel group, which is obtained from the training framework. Mixtera currently allows to express static filter operations on properties, and static as well as dynamic mixtures across all properties (G4). For more details on supported mixture types, we refer to Section 5.2.2.

Dataflow. Mixtera follows a server-client model. As shown in Figure 3, the server runs on a node and each training node runs client instances. Users first register samples at the server to populate the metadata database. They then can submit queries. A query is executed at the server in two phases. First, Mixtera applies static filters from the query (e.g., English-only) to obtain all samples eligible for training. Second, during training, the server distributes *chunks* of that query result to the client(s), which specify which samples to train on. The server ensures that the chunks are distributed correctly, i.e., tensor- and pipeline parallel stages receive the same input data. The server generates chunks according to the current mixture, i.e., it iteratively takes samples from the query result generated in the first phase of query execution, such that data in the chunk follow the current mixture. As an iterable data loader, Mixtera faces the challenges of determinism and checkpointing. We address this by shuffling based on the query and support to load/store the query state.

Chunks. Mixtera does not store the sample payloads, but rather only the metadata of each sample. During training, the nodes receive *chunks*. A chunk is a fixed-size collection of pointers to samples in files. They tell the client which samples which file to load and train on (e.g., sample 10 in file `wikipedia.jsonl.zst`). The files can be local, in a distributed filesystem, or cloud storage.

Storing and distributing pointers to samples instead has several advantages. First, users can store data in their locations of choice (e.g., an object store, or an distributed filesystem). Chunks are independent of the filesystem structure (G1). Second, it allows

Mixtera to support dynamic mixtures (G4), as the data composition of chunks can change over time. Third, the pointer-model avoids creating a data fetching bottleneck at the Mixtera server. The server only creates chunks and each client fetches the data they need. Fourth, we avoid a lock-in effect on Mixtera and allow for easy adoption on existing data collections (G2). Last, we natively support other modalities like images, which would not be straightforward to ingest at scale into a database.

Mixtures. Users can use any subset of sample properties to define a mixture. Users can even change the properties that they mix on during training with dynamic mixture schedules. To handle this, Mixtera implements a `MixtureKey` abstraction that allows users to describe which samples they want to use. The keys also implement flexible property matching to define which samples to use during chunk generation (Section 5.2.2).

High-throughput data fetching. The challenge of using chunks is that we need to handle suboptimal data layout. Files may have arbitrary property distributions, or might follow a partial structure (e.g., a file only contains Wikipedia data, but the languages are distributed randomly). We cannot re-organize data within the existing files. Formats like `jsonl` were not built with random access in mind, but chunks force clients to load individual samples from files. To avoid data stalls (G3), we implement chunk generation to use subsequent sample ranges in files if the samples have the same properties and engineer Mixtera's implementation to fetch those ranges as efficiently as possible (Section 5.3).

Open source. Mixtera comes as a Python package that provides the entrypoint for the server and abstractions for the client. Its codebase, consisting of approximately 11 k lines of Python and C++ (excluding tests), is open-source[1]. It is rigorously tested with a full set of unit and integration tests. We are continuously working on enhancing the system and welcome contributions.

5 IMPLEMENTATION

We explain how sample metadata is managed (Section 5.1) at the server, how it executes queries and creates chunks (Section 5.2), how those chunks are parsed at the client (Section 5.3), and give details on Mixtera's integration into training frameworks (Section 5.4).

5.1 Sample Management

Mixtera manages samples in a `MixteraDataCollection` (MDC). It keeps track of all samples and their properties. The MDC leverages DuckDB [51] as the robust and flexible metadata management system underlying it. Every registered sample has properties according to a schema defined by a `MetadataParser`, e.g., language. Representing the structure of training data, Mixtera initializes the database with three tables for source datasets, files, and samples. The source dataset can also be ignored and stored as a schema property instead, to support collections such as The Pile [19] where data from different datasets is contained within the same files.

`MetadataParsers`. A `MetadataParser` is a class that defines a schema for a data collection and, given a sample, extracts and returns the properties according to the schema. A schema is a list of properties which have a type (e.g., string or enum), a nullable field, and a multiple field, describing whether a single sample can take multiple values for this properties, e.g., multiple languages. Mixtera maps this Python schema to a proper database schema, comes with a set of pre-defined parsers for common datasets, and enables users to define custom parsers. For example, there is a parser for The Pile [19]. Every sample has the property `pile_set_name` describing the source dataset. This is a non-nullable, non-multiple, enum property, since we know the limited set of possible values.

Data types. Based on the parser, Mixtera adjusts the DuckDB sample table. Properties marked as multiple get mapped to DuckDB's list type, e.g., a multiple string property gets mapped to `VARCHAR[]`. Enum properties are appropriately inserted into DuckDB. Using enum instead of strings for properties where all values are known beforehand speeds up query execution in DuckDB.

Insertion. When inserting (registering) data, the MDC accumulates all relevant files, and prepares the DuckDB table schema. Then, all files are inserted into the files table, and a pool of workers in parallel processes all files using the `MetadataParser` to extract the sample metadata. We then aggregrate the worker results and insert it into the database, as DuckDB does not support insertion from multiple Python process workers. We found that converting the worker results to columnar pyarrow in-memory tables and then inserting into the samples table has the highest throughput.

5.2 Server-Side Query Execution

After receiving a query from the client, the server executes it in two phases. The first phase is performed via the MDC DuckDB-abstraction and applies static filters to identify all potentially relevant samples, e.g., all non-English samples are filtered out, and groups consecutive samples into intervals. The second phase constructs a specialized data structure called `ChunkerIndex` that enables efficient, mixture-aware chunk generation.

5.2.1 SQL generation and interval detection. After receiving an object representation of the query (c.f. Figure 4), similar to ORM frameworks like sqlalchemy, the Mixtera server generates a base SQL query from this object. This query returns a table in which each row represents a sample that the user is interested in. Mixtera ensures that the generated SQL matches the MDC's table schema, e.g., whether a property can have multiple values or not.

A key challenge for Mixtera is efficient random access to samples within files. File formats like jsonl or parquet are optimized for sequential reading rather than random access. To address this, Mixtera implements an interval-based approach: the server wraps the base filtering query in an outer query that identifies continuous ranges of samples sharing identical properties within the same file. Consider the following example result of a base filtering query:

Sample ID	File ID	Language	License
1	1	JavaScript	MIT
2	1	JavaScript	MIT
3	1	JavaScript	MIT
4	1	Python	Apache
5	1	Python	Apache
1	2	Python	Apache

Instead of treating these as six individual samples, Mixtera identifies three intervals:

- Interval 1: Samples 1-3 (File 1, JavaScript, MIT)
- Interval 2: Samples 4-5 (File 1, Python, Apache)

[1] Available at https://github.com/eth-easl/mixtera.

- Interval 3: Sample 1 (File 2, Python, Apache)

Even though samples 4-5 (file 1) and 1 (file 2) share the same properties (Python, Apache), they are in different files and thus form separate intervals. The primary key is formed by the sample and file ID together. Mixtera constructs a SQL query that processes the data in multiple stages:

(1) First, Mixtera establishes a Common Table Expression (CTE) named `base_data` that contains the filtered samples:

```
WITH base_data AS (
    -- Our generated base filtering query here, e.g.,
    SELECT * FROM samples WHERE license = 'MIT'),
```

(2) Next, Mixtera identifies breaks in the sample sequence using window functions. The `grouped_samples` CTE calculates the difference between consecutive sample IDs within groups sharing the same properties:

```
grouped_samples AS (
    SELECT *, sample_id - LAG(sample_id, 1, sample_id)
    OVER (PARTITION BY file_id, lang, license
          ORDER BY sample_id) AS diff
    FROM base_data),
```

Here, a `diff` value of 1 indicates consecutive samples, while any other value indicates a break in the sequence.

(3) The `intervals` CTE then groups the sequences into intervals:

```
intervals AS (
    SELECT file_id, lang, license,
        SUM(CASE WHEN diff != 1 THEN 1 ELSE 0 END)
            OVER (PARTITION BY file_id, lang, license
                  ORDER BY sample_id) AS group_id,
        MIN(sample_id) as int_strt, MAX(sample_id)+1 as int_end
    FROM grouped_samples
    GROUP BY file_id, lang, license, diff, sample_id)
```

The `group_id` is incremented when there is a break in the sequence, creating unique identifiers for each interval.

(4) Finally, Mixtera aggregates the results to get the final intervals:

```
SELECT file_id, lang, license, group_id,
  MIN(int_strt) as interval_start, MAX(int_end) as interval_end
FROM intervals
GROUP BY file_id, lang, license, group_id
ORDER BY file_id, interval_start;
```

While we initially implemented the interval aggregation within Python, letting DuckDB optimize and parallelize the calculation of the interval is more efficient. This optimization can only improve I/O if samples within files are clustered by properties and not randomly distributed. For example, if a user has a file structure based on the source dataset and mixes based on that, Mixtera can heavily leverage the range optimization and just read, e.g., all Wikipedia samples from the sample file. If the user decides to now mix on language as well, there might only be local clusters within the files. Since Mixtera is read-only by design, it cannot re-shuffle data.

5.2.2 Chunk generation. The chunk generation algorithm is based on the `ChunkerIndex` data structure, which organizes sample ranges by their properties. This data structure is generated based on the output from the interval query. Before explaining its construction, we first introduce the key abstraction that enables flexible property matching: `MixtureKey`.

`MixtureKey` abstraction. A `MixtureKey` represents a set of properties and their values, e.g., language being English and license being MIT. A property in a key can have multiple values. A key *matches* another if their lists of values intersect for all properties they share. This matching is crucial because the resulting interval table from DuckDB contains the full cross-product of all properties—a sample might have values for language, license, size, topic, and more—while a mixture specification typically considers only a subset of these properties. For example, a mixture might specify `language:English` to select English text regardless of license, while the underlying samples are stored with full property information (e.g., `language:English,German;license:CC` and `language:English;license:MIT`). The flexible matching enables finding all samples that satisfy certain properties while ignoring irrelevant attributes they might have. It also allows us to define mixtures on *multiple properties* with *multiple values*, instead of being limited to a single property (c.f. Section 3). To ensure deterministic behavior, we implement a total ordering over keys based on the number of properties, property names, and their values. We sometimes refer to a specific `MixtureKey` as a *domain*, e.g., the key for `language:English` defines the domain for English samples.

The `ChunkerIndex`. The `ChunkerIndex` is a nested mapping from `MixtureKeys` to sample locations. We refer to keys in the `ChunkerIndex` as *component keys*. For each unique combination of all available properties and values, the index maps to dataset IDs, which in turn map to file IDs and finally to sorted lists of intervals of samples which share the properties and values as specified in the `MixtureKey`. While the index maintains the complete property information of samples, the matching semantics allows us to efficiently query arbitrary subsets of these properties. Consider a simplified example where we represent `MixtureKeys` as strings. A fragment of the `ChunkerIndex` might look like:

```
{ "language:JavaScript,HTML;license:MIT": {
    ds_1: {
      file_1: [(1,4), (10,15)],  # half (right) open ranges
      file_2: [(1,2)]
    }},
  "language:Python;license:Apache": { ds_1: { file_1: [(1,2)] }}}
```

In this example, a query for `language:JavaScript` would match the first key despite it having the additional license property and two assigned languages. This demonstrates how the `MixtureKey` matching allow to work with the full property/value cross-product in the index while supporting mixtures on subsets of properties.

Building the `ChunkerIndex`. The index is built in parallel in a C++ extension, processing the interval table from DuckDB provided in Apache Arrow format. Operating on the Arrow table in Python would be too slow due to Global Interpreter Lock (GIL) constraints. Using `multiprocessing` to circumvent the GIL would require expensive pickling of nested dictionaries. Our C++ implementation uses multithreading and only acquires the GIL at the end when creating the final Python result object.

Each C++ worker thread maintains a local index for a subset of the data. For each row (interval), each worker constructs a C++-representation of the `MixtureKey`, inserts the interval into its local index under this key, maintaining sorted order within each file's

Algorithm 1: Chunk generation algorithm. Some early exits and details are omitted for readability.

```
1  Initialize remaining_counts from mixture;
2  chunk ← ∅;
3  progress ← true;
4  while ∃key : remaining_counts[key] > 0 and progress do
5      progress ← false;
6      foreach mixture_key in remaining_counts do
7          foreach component_key in chunker_index do
8              if mixture_key matches component_key then
9                  Take up to remaining_counts[mixture_key]
                     samples from
                     chunker_index[component_key];
10                 if got > 0 samples then
11                     Add samples to chunk;
12                     Update remaining_counts;
13                     progress ← true;
14         if remaining_counts[key] > 0 and is best-effort then
15             Redistribute remaining counts to other keys;
16 if ∀key : remaining_counts[key] = 0 then
17     return chunk;
```

interval list. After parallel processing, the local indices are merged, combining interval lists while preserving their sorted order. In the end, we convert the index to Python objects, which often is the most expensive operation of this process.

Chunk generation. A mixture in Mixtera is represented as a mapping from `MixtureKeys` to their target proportions within a chunk. Given a chunk size that describes how many samples each chunk should contain, these proportions are converted to absolute counts following the largest remainders quota method. Users can set a mixture to be *strict*, requiring exact proportions, or *best-effort*, allowing to continue to generate chunks even if the mixture cannot be exactly fulfilled. The chunk generation algorithm (Algorithm 1) iteratively constructs chunks that conform to the current mixture. This algorithm supports dynamic mixture, as the mixture can be changed between chunks.

Similar to the `ChunkerIndex`, a chunk is also structured as a hierarchical dict with keys and files pointing to lists of ranges. This chunk can be parsed on a training node by loading the sample ranges from the files (Section 5.3). While the `ChunkerIndex` uses all properties/values as `MixtureKeys` (component keys), a chunk's keys are identical to the mixture's keys, which might only contain a subset of properties.

For each key in the mixture, the algorithm keeps track of how many samples we still need to put into the chunk that is currently being generated (`remaining_sizes`). For each key in the mixture (line 6), it checks whether it matches a component key in the chunker index (lines 7-8). If we find a match, we try and obtain samples (ranges) from the `ChunkerIndex` for this component key (lines 9+).

A call to obtain samples for a component key can return fewer samples than requested, e.g., if we are looking for JavaScript data and we need 5 samples, but we only have 3 JavaScript/MIT licensed samples, the according component key can only return 3 samples. Requesting n samples is implemented as requesting m *ranges* (from potentially multiple files) such that the overall number of samples in the returned list of ranges is $\leq n$. The lists of ranges per file are merged into the existing sorted list of ranges. Mixtera's implementation uses Python generators that yield ranges of samples and accept the number of samples needed as input through the Python generator's send mechanism. This hides the complexity of the *take samples* operation. It also allows for efficient, stateful iteration over the available ranges while maintaining control over how many samples are returned in each request. There is one generator per component key that returns the ranges containing N samples based on the `ChunkerIndex` but ensures ranges are split such that never too much data is returned.

If after traversing all component keys we did not find sufficient samples for a key in the mixture, in strict mode, chunk generation fails. In best-effort mode, the algorithm redistributes any unfulfilled counts to the remaining mixture components proportionally to their original ratios. For example, if we need 100 JavaScript samples but only find 80, the remaining 20 samples would be proportionally distributed among other components. To avoid infinite loops, we only distribute samples to keys on which we were able to find any samples in the last iteration.

Mixtera's implementation of this algorithm ensures determinism because (1) the mixture's keys are processed in a consistent order and (2) when multiple component keys match a mixture key, they are considered in a deterministic order based on a seeded shuffle of all possible keys. This approach ensures that identical queries with identical mixtures always produce identical chunks, important for training debugging and reproducibility [15, 27, 50, 64, 79].

Mixtera also supports an alternative, *arbitrary chunk* generation algorithm. It does not require any specified mixture and builds up chunks by iterating over the component keys, moving on to the next one when one key is depleted. This can be useful if the data is completely property-free and guarantees maximum throughput as all ranges are consecutive.

Mixture types. All mixture classes implemented in Mixtera share a common interface that converts their specifications into a mapping from `MixtureKeys` to sample counts per chunk, used by the chunk generation algorithm:

- Static Mixture: Users explicitly specify fixed proportions for different property combinations (Figure 4). This supports arbitrary properties and is not limited by, e.g., directory boundaries.
- Inferring Mixture: Automatically derives mixture proportions from the data distribution in the query result: This is useful when users want to maintain the natural distribution of properties.
- Hierarchical Mixture: An advanced static mixture that allows to specify nested property relationships. For example, users can define that 50 % of the data should be legal texts, and within that, 60 % should be in English and 40% in French. Mixtera automatically flattens this hierarchy into appropriate `MixtureKeys`.
- Mixture Schedule: A "meta mixture" that allows for temporal changes in mixture composition by defining a sequence of mixtures that activate at specific training steps. This enables curriculum learning with predefined schedules.

– Dynamic Mixture: Allows adaptation of mixture proportions during training based on feedback (e.g., loss) from the model. If an algorithm is already supported by Mixtera (e.g., ADO), it can be used directly.

Chunk distribution. In distributed training, it is important to guarantee that all nodes within the same data parallel group operate on exactly the same input tensors. Mixtera's `ChunkDistributor` wraps around the chunk generation component, and hands out chunks correctly to the training nodes, i.e., the same chunks in the same order to nodes within the same group, and different chunks to nodes in different groups for data parallelism. To this end, the clients need to register at the server with their respective node and group identifiers. To avoid redundant serialization overhead, the distributor caches chunks in serialized form until all nodes in a group have received them.

5.2.3 Networking. We implement a TCP-based client-server protocol. The server uses Python's asyncio framework to handle multiple concurrent client connections. The protocol is message-based, with each message consisting of a task identifier followed by task-specific payload data. Tasks for example include the execution of a query or sending out a new chunk to a client. To handle network issues gracefully, the client implementation includes automatic reconnection with exponential backoff and configurable timeouts. Larger data transfers, such as chunk transmission, use efficient streaming to avoid memory issues.

5.3 Client-Side Reading

Mixtera's client-side abstractions offer a generator that yields data samples for a query at the server. This generator internally follows a two-level nested iteration pattern: an outer iteration over chunks and an inner iteration over samples within each chunk. The outer iteration hides the complexity of network transfer and chunk generation, while the inner iteration hides the complexity of going from pointers in the chunk to actual samples. We discuss this inner step, i.e., how, given a chunk, we yield sample payloads.

Processing modes. A chunk can be processed in three mixture processing modes with different trade-offs. The modes influence in what order samples are yielded, i.e., in what granularity the mixture is guaranteed, and whether string samples or tokenized sequences are yielded. All modes begin by instantiating one *active iterator* per property combination. These iterators traverse files and ranges for their respective properties in a randomized order while maintaining sequential reading within consecutive ranges for I/O efficiency.

Overall mixture mode. This mode processes active iterators in a randomized round-robin fashion until depletion. This ensures the mixture ratio is maintained at the chunk level.

Window mixture mode. This mode guarantees the mixture on a window smaller than the chunk size. Similarly to chunk generation, we determine how many samples per property we yield within a window. We then go through the properties in a randomized, round robin fashion until a window has been yielded, and start again. This mode can operate in best-effort (continues after mixture cannot be guaranteed) or strict mode (stops at the first window where the mixture cannot be maintained). In strict mode, the number of overall samples yielded from the chunk might be smaller than the chunk size.

Tokenized mixture mode. This mode extends the strict window mode. It wraps the active iterators with a *tokenizing iterator* that takes the incoming string samples, tokenizes them, and yields tokenized samples (integer lists) with the correct sequence length. By setting the window size equal to the chunk size we guarantee that each chunk at least yields one window of *tokenized samples*. This mode is important for handling imbalanced collection with significantly varying text lengths. For example, in the The Pile [19], an average sample from the `Books3` domain is much longer than one from the `PubMed Abstracts` domain.

Without tokenization, one string sample may correspond to potentially order-of-magnitudes more tokenized samples, which can heavily disturb the actual mixture seen during training. While this mode ensures precise mixture ratios at the token level, it may result in partial utilization of longer samples. This is an inherent issue of unbalanced datasets, and while Mixtera provides flexibility to handle it, the best approach is to process the datasets such that samples are (roughly) of similar size.

File reading. The active iterators wrapped by the processing iterators shuffle the file order but maintain sequential reading within files. This acknowledges that formats like `jsonl` and `parquet` are optimized for sequential rather than random access, and enables us to linearly iterate through the sorted lists of ranges per file. The complexity of reading different file formats internally is hidden by abstractions for each format. Using the `xopen` library we support both compressed and uncompressed `jsonl`. We optimize the reading of `parquet` files by calculating and loading only the relevant the row groups, and build upon pyarrow's `parquet`-batched-reading implementation. WebDatasets is the only format supporting random access to samples, and we implement support using the `wids` library. The format also gives us the option to store text-image pairs, as it can contain different modalities. To mitigate latency from initial file operations that we observed on distributed filesystems, Mixtera employs a prefetching iterator that uses background threads to hide file opening latency.

Determinism. All random operations are seeded based on the current chunk, ensuring identical behavior across nodes. Combined with the server-side chunk generation and distribution, this guarantees that *all clients within a data parallel group yield exactly the same samples in exactly the same order.* This property is crucial for both reproducibility across runs and correctness in distributed training. We validated this using a suite of integration tests as well as the dataloader verification test provided by nanotron.

Dataset abstractions. Training frameworks typically require specific dataset interfaces that supports `multiprocessing` with worker processes. Besides a general purpose interface, Mixtera offers a class extending torch's `IterableDataset`, and a class compatible with the huggingface API. Each worker process at each node operates on its own chunk.

5.4 Framework Integration

Mixtera integrates into the training framework for checkpointing and transferring model feedback (e.g., per-domain loss).

Checkpointing. Mixtera's API offers a function to be called on checkpoint. In order to restore from checkpoint, we need to know (i) which chunks have been handed out to which nodes and (ii) which

Table 2: Model configurations.

	Hid. Dim.	Interm. Dim.	KV-Hds.	Q-Hds.	Layers	RoPE-θ
162M	768	2 048	12	12	12	10 000
1.6B	2 048	5 464	16	16	24	10 000
3.6B	3 072	8 192	8	24	28	500 000

data loader worker processes have yielded how many samples for each node. Mixtera implements (i) using the `ChunkDistributor`, which caches the query and the current state on checkpoint and is able to restore the in-memory state of the iterators for chunk generation based on this information. For (ii) the `MixteraTorchDataset` uses a shared memory segment to share with the main training process the status of the data loader workers. When we restore a checkpoint, we restore the state at the server, hand out the last chunks to each worker that they were working on, and then at the workers discard the previously yielded samples.

Training feedback. Dynamic mixing algorithms require a loss per property domain. Mixtera offers a simple function to forward this information to the server. Users still need to adjust the training framework. The loss implementation needs to be adjusted s.t. it is not immediately reduced but stored per domain. The per-domain losses along all training nodes need to be synchronized, e.g., via all-reduce, before passing it to Mixtera.

6 EVALUATION

We evaluate Mixtera to answer the following questions:

(1) How can we integrate dynamic mixing algorithms into Mixtera and what role do mixtures play for model accuracy?
(2) How does Mixtera's throughput compare to other data loaders and how well does it scale?

We explore the first question in a dynamic mixture case study on LLMs (Section 6.1) and a static mixture case study for VLMs (Section 6.2); the second question is explored in throughput benchmarks (Section 6.3). We conduct our experiments on the Clariden partition of the Alps supercomputer of the Swiss National Supercomputer Centre. Clariden consists of HPE Cray Supercomputing EX254n blades, each hosting two Quad GH200 nodes. Each node contains 4 interconnected groups of a Grace CPU with 72 cores, 128 GB DRAM, and a H100 Hopper GPU with 96 GB of HBM. The nodes are connected using 200 Gb/s HPE Slingshot interconnect. We refer to Fusco et al. [18] for a more detailed analysis of the Alps supercomputer. The machines run Ubuntu Server 24.01 LTS with kernel 5.14.21, and we build upon the NVIDIA NGC 25.01 container with Python 3.12, a nightly build of PyTorch 2.6, NVIDIA driver 550.54.15 and CUDA 12.8. We add support for Mixtera and other data loaders on our fork of torchtitan (commit `d989842`)[2] [32]. Torchtitan is part of the Pytorch ecosystem and straightforward to set up. We use Llama3-like models whose configurations can be found in Table 2. The 162M and 1.6B models are based on Jiang et al. [26], while 3.6B follows Meta's Llama 3.2 model. They do not have the same parameter count as torchtitan does not tie the weight embeddings. Our training and benchmarking data is based on The Pile [19]. We heuristically split long samples with more than 3000 words, with max. 10 k samples per file.

[2]Available at https://github.com/eth-easl/torchtitan-mixtera.

6.1 Dynamic Mixing using ADO

We first demonstrate how to implement dynamic mixing algorithms in Mixtera, taking the ADO algorithm (Section 2.1.1) as an example. We show how dynamic mixing can improve model performance during pre-training. Dynamic mixing algorithms are an active area of research [3, 11, 12, 26, 31] and Mixtera facilitates exploration.

Mixtera implementation. The original ADO implementation updates the current mixture π after every step and samples the next batch based on this distribution. The distribution is updated based on the per-domain loss from the previous step. This does not fully match Mixtera, which can only use a new mixture when generating a new chunk. Each chunk then may yield several batches of data with the same mixture. We still send the per-domain losses on every training step at the client to update ADO's internal state at the server. Whenever a new chunk is generated, the current mixture π from ADO is queried (Algorithm 1), and the server generates a chunk according to that mixture. This introduces some slack since we do not use a new mixture on every batch. However, due to the stochastic nature of sampling, we do not find this to be an issue.

The only change needed in training clients is the implementation of a per-domain loss. For this, the loss function (e.g., cross-entropy) needs to be called without reduction, which gives a loss *per token*. As Mixtera provides which token belongs to which domain, we can aggregate the losses *per domain*. We then perform an all-reduce operation across all training nodes to get the global per-domain losses and send this to the server.

While the original implementation of ADO is fully tied to their training framework and data setup, Mixtera is agnostic to the training framework. While developing, we switched from nanotron [24] to torchtitan [32]. No changes in Mixtera were required. Only the per-domain loss module at the training framework is required. This showcases the benefit of having a system like Mixtera that decouples the mixing from the training framework.

Training setup. We base our setup on Jiang et al. [26]. In addition to their 162M and 1.6B architectures, to test how ADO scales to the latest model architecture, we try a 3.6B model (Table 2) following `Llama-3.2-3B` from Meta, including its tokenizer. The 162M and 1.6B models use the `EleutherAI/GPT-NeoX-20B` tokenizer. All models use a sequence length of 2048. For ADO, we follow the codebase and discard the first 500 steps for fitting the scaling laws, start with fitting the scaling laws at step 1 000, and then re-fit the laws every 1 000 steps with a subsampling frequency of 10. We also follow the codebase and "use the same step size" for all domains. Instead of using the count of how often a domain has been sampled to fit the scaling laws like in the paper, we follow the ADO codebase and average the total sample counts evenly across all domains. We train all models with non-strict token-level mixtures, a learning rate of 0.001 with a linear warmup for 500 steps and linear decay until the final step, and the AdamW optimizer, for 30 000 steps. For 162M/1.6B, we use 64 GPUs with a microbatch size of 32, and for 3.6B we use 128 GPUs with a microbatch size of 16, resulting in a global batch size of 2048 and total 125 B tokens. As mixtures, we test ADO, the default weights as in the DoReMi paper [69], and on 162M/1.6B, the "natural" weights by Jiang et al. [26] which are optimized for the NeoX tokenizer.

Table 3: Task performance and perplexities across models, checkpoints, and mixtures. ↑/↓ indicate higher/lower is better.

Model	Steps	Mixture	HellaSwag ↑	WinoGrande ↑	ARC-E ↑	ARC-C ↑	LogiQA2 ↑	Lambada (OAI) ↑	OpenBookQA ↑	PIQA ↑	SlimP. PPL ↓	Pile PPL ↓
162M	15 k	ADO	**0.28**	0.49	0.43	**0.20**	**0.23**	**0.29**	**0.19**	**0.61**	58.19	**66.88**
		Default	**0.28**	**0.52**	**0.44**	0.19	**0.23**	**0.29**	0.18	**0.61**	59.13	69.52
		Natural	**0.28**	**0.52**	0.41	0.18	**0.23**	0.26	0.16	**0.61**	**57.54**	69.85
	30 k	ADO	**0.29**	0.50	**0.44**	0.19	0.23	0.31	**0.17**	0.62	**52.62**	**61.82**
		Default	0.28	0.52	**0.44**	**0.20**	0.23	**0.34**	0.16	0.62	53.55	64.39
		Natural	**0.29**	**0.54**	**0.44**	0.19	**0.24**	0.31	0.16	**0.63**	52.78	65.03
1B	15 k	ADO	**0.37**	**0.54**	0.51	**0.23**	**0.25**	**0.52**	**0.20**	**0.70**	30.07	**32.20**
		Default	0.35	**0.54**	**0.52**	**0.23**	0.22	0.47	0.17	0.68	**29.34**	33.02
		Natural	0.35	**0.54**	**0.52**	**0.23**	0.23	0.50	0.17	0.68	31.31	34.40
	30 k	ADO	**0.40**	0.55	0.55	**0.26**	**0.23**	0.54	**0.23**	**0.71**	25.89	29.05
		Default	0.38	**0.56**	**0.56**	0.24	0.22	0.56	0.21	0.70	**25.10**	**28.65**
		Natural	0.38	**0.56**	0.55	0.25	0.22	**0.57**	0.21	0.69	26.55	29.83
3B	15 k	ADO	**0.42**	**0.56**	**0.57**	**0.25**	**0.24**	**0.59**	**0.21**	**0.71**	**25.78**	**23.00**
		Default	0.38	0.54	0.52	**0.25**	**0.24**	0.50	0.20	0.68	26.21	24.73
	30 k	ADO	**0.45**	**0.59**	**0.60**	**0.28**	**0.24**	**0.59**	**0.24**	**0.73**	**22.22**	**20.52**
		Default	0.42	0.56	0.59	0.26	**0.24**	0.57	0.23	0.70	22.52	21.79

Evaluation metrics. We follow Jiang et al. [26] and report both downstream task performance as well as perplexity. For downstream tasks, we report performance on HellaSwag [75], WinoGrande [54], ARC-Easy and ARC-Challenge [13], LogiQA2 [35], Lambada OpenAI [47], OpenBookQA [40], and PIQA [7]. For perplexity, we report the average unweighted token perplexity on (i) the validation set of The Pile [19], and on (ii) SlimPajama [59] as a dataset we did not train on. We collect all metrics using EleutherAI's lm-eval-harness [20], and use the unnormalized accuracy.

ADO algorithm performance overview. Table 3 shows the performance of all models and mixtures for a checkpoint after 15 k and 30 k steps. We mark in bold the best value within a model/step group. In contrast to Jiang et al. [26], we find that on the 162M model, the default mixture is quite competitive across benchmarks. For the final checkpoint (30 k), ADO achieves the best (lowest) perplexity on both SlimPajama and The Pile, but it does not consistently beat the static mixtures as reported in the ADO paper. However, analysis of the textual outputs reveals that the quality of generated text from this small model is limited across all configurations.

For the larger 1B model, ADO is particulary strong on the intermediate checkpoint, which shows an improvement in time-to-accuracy. For the final checkpoint, on some benchmarks, the static mixtures perform on-par or marginally better than ADO, but on others, such as HellaSwag, ADO clearly beats the static mixtures.

On the 3B model, ADO beats the static default mixture on both checkpoints across all benchmarks. While Jiang et al. [26] report that ADO performs best on the smaller scale model, we find that ADO performs better on larger models. Since ADO's official repository is tightly coupled with the training framework, even after corresponding with the authors, we were not able to identify the root cause of this, partly also because their code is bound to training on specific cloud instances. This shows that a common codebase for data mixing decoupled from the training framework is beneficial and help developers integrate the latest algorithms into their setups (Section 3).

Performance over time. To demonstrate how different tasks behave over the training, we show the performance of the 1B model on HellaSwag, OpenBookQA, and ARC-Easy for all training checkpoints in Figure 5. Every benchmark exhibits different behavior. For HellaSwag, we can clearly see that ADO consistently increases its margin over the static mixtures. For OpenBookQA, ADO performs well especially in the intermediate checkpoints, and the static mixtures greatly improve towards the end. For ARC-Easy, all mixtures behave similarly. This motivates future research on data mixing using Mixtera. For example, we might be able to use intermediate evaluations instead of loss to dynamically adjust the mixture.

Mixture over time. We showcase the mixture over time for the six largest domains in Figure 6. Around step 21 k, we run out of data for the Books3 domain, and Mixtera's best-effort algorithm re-distributes the weight proportionally to the other domains. This happens because The Pile's samples are very imbalanced in size, i.e., the average sample in Books3 has around 538 KiB, while the average Pile-CC sample has 4 KiB. While we split long samples heuristically, there still is some waste due to token-level mixtures. Unlike Jiang et al. [26], we do not observe that other models/tokenizers lead to different mixtures. Notably, the more parameters, the higher the weight of Books3, and the lower the weight of PubMed Central. On the 3.6B model, OpenWebText2's weight even surpasses PubMed Central's weight before step 5 000.

Takeaways. Dynamic mixture algorithms can improve model accuracy. ADO scales beyond the 1.6B model tested in the original paper, beating the default weights on the 3.6B model on all benchmarks. Our experiments also demonstrate that a synchronous algorithm, which uses a new mixture at each training step, adapts to Mixtera's chunking system. We observe some discrepancies on the 162M model, which highlights the importance of having a common, open-source data mixing platform like Mixtera that facilities debugging and reproducibility.

6.2 Multimodal LLaVA Finetuning

We are not aware of prior work on the impact of data mixtures on VLMs. To showcase Mixtera's multimodal capabilities, we evaluate the impact of static mixtures for finetuning a LLaVA-style model [34]. The LLaVA framework trains an adapter between a pre-trained image encoder and a pre-trained LLM, and then fine-tunes the adapter and LLM on visual instruction-following data.

Training setup. We base our training setup on the TinyLlaVA Factory codebase by Jia et al. [25]. We rely on a recipe from the

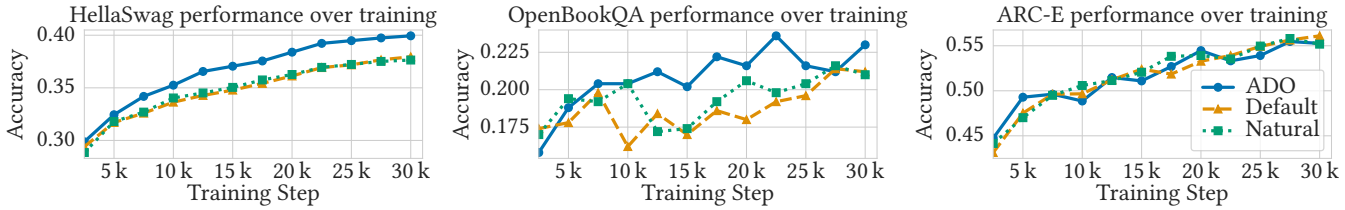


Figure 5: Performance of the 1B model on HellaSwag, OpenBookQA, and ARC-Easy, measured every 2 500 steps.

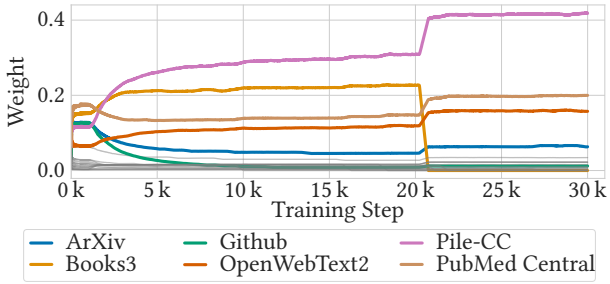


Figure 6: Mixture for the 1B model for the 6 largest domains.

Factory and use `google/siglip-so400m-patch14-384` [2] as the vision encoder, `TinyLlama/TinyLlama-1.1B-Chat-v1.0` [76] as the LLM, and a 2-layer MLP as the adapter [62]. We train all models on one GH200 node with 4 data parallel GPUs. For pre-training, we use a global batch size of 512 with a learning rate of 0.001, and for finetuning use a global batch size of 128 with a learning rate of 0.00002. We use a chunk size of 256, a cosine learning rate scheduler and the Adam optimizer. We follow Liu et al. [34, 61] and pre-train the adapter on a 558 k subset of the LAION-CC-SBU dataset with BLIP captions. For finetuning, we follow the TinyLLava Factory "LLaVA dataset" and use 665 k samples from six datasets (COCO [33], GQA [23], OCR-VQA [41], TextVQA [58], and VisualGenome (VG) [28], and LLaVA's text-only SFT annotations [34]) [65]. We pre-train the adapter once, and then vary the proportions of the finetuning datasets. We randomly generate 256 mixtures for finetuning. Since the number of datapoints in comparison to LLM training is small, we use a best-effort mixture and ensure we go through all samples exactly once. All models see the same data, but in different order.

Benchmarks. We evaluate the models on the GQA [23], SQA-IMG [37], TextVQA [58], POPE [30], MME [17], and MMMU [73] benchmarks. We collect all metrics using TinyLLaVA Factory.

Results. In Table 4, we show the results reported by Jia et al. [25], the results we obtain using the inferring mixture (Section 5.2.2), and three mixtures out of the generated mixtures that perform well. While the inferring mixture does not achieve their reported results, this could be either due to different data dynamics in their training, or due to a different evaluation setup we cannot reproduce as their model weights are not public. Nevertheless, in particular on MME/MMMU/POPE, the mixtures outperform the baseline. Mix. #252 is weighted towards TextVQA (29.1 %) and OCR-VQA (19.5 %), with equal proportions of COCO (21.9%) and VG (21.9 %); GQA (4.0 %) and SFT annotations (3.6 %) contribute minimally. Mix. #107 places a large emphasis on SFT annotations (28.9 %) and VG (27.4 %), followed by COCO (25.5 %); GQA (5.3 %) and TextVQA (5.1 %) have lower representation, while OCR-VQA (7.8 %) remains a minor component. Mix. #155 prioritizes OCR-VQA (30.3 %) and TextVQA (25.7 %), and SFT annotations (22.7 %); VG (13.2 %) and COCO (6.9 %) are underrepresented, while GQA (1.2 %) is least utilized. Overall, despite seeing the same samples globally, the mixtures play a big role for model accuracy. Exploring mixtures for pre-training of VLMs and adapting dynamic mixtures to VLMs are promising future work directions.

Table 4: VLM scores for hand-picked mixtures.

Model	SQA-IMG	TxtVQA	GQA	MME	MMMU	POPE
Jia et al. [25]	64.0	49.6	58.6	1256.5	28.3	86.3
Infer. Mix.	55.88	37.92	54.63	1238.76	**29.4**	**86.6**
Mix. #252	63.01	43.87	56.14	**1268.05**	**30.4**	85.7
Mix. #107	58.50	45.18	58.36	**1283.62**	**30.1**	85.54
Mix. #155	57.14	42.37	57.14	**1290.06**	**29.3**	**86.63**

6.3 Throughput Benchmarks

We now focus on throughput and compare Mixtera with other data loaders across various configurations. Mixtera's goal is to avoid data stalls, i.e., training throughput should not be reduced because we are waiting for data [29, 42, 43, 77]. Hence, we measure throughput in tokens per second in actual workloads. Note that *smaller models* and *more data parallelism* increase the pressure on the data loader, while larger models reduce the pressure. If a data loader can sustain training small models at scale on a high-end platform like the GH200, its performance is sufficient for other scenarios as well.

Besides Mixtera with chunk sizes of 512, 1024, and 2048, we benchmark the iterable `HuggingFaceDataset` by Torchtitan (Hf-Iter), a mapped version (Hf-Map), the Mosaic StreamingDataset [63] (Mosaic). The difference between Hf-Iter and Hf-Map is that similar to Mixtera, Hf-Iter loads and tokenizes the data on the fly, while Hf-Map preprocesses all data, including tokenization. We evaluate throughput on the 162M model since larger models only lead to lower throughput. We always use FSDP since Torchtitan only enables `bfloat16` training if sharding is used. We activate compilation, disable activation checkpointing, and use fused AdamW. We measure throughput for 30 steps, discarding the first step. We repeat all measurements three times, i.e., in total we have 3x30 steps. We use the huggingface `EleutherAI/gpt-neox-20b` tokenizer for all data loaders. We store all data on the `Iopsstor` SSD-backed Lustre DFS in the Alps cluster.

Single-node. We train on a single GH200 node with 4 GPUs. We use 2 data parallel replicates and shards. In Figure 7 we show the throughput for the data loaders depending on the number of data workers. We test up to 16 workers, where 0 workers indicate that data is loaded in the same process as the main training loop. We only show results up to 4 workers due as throughput does not

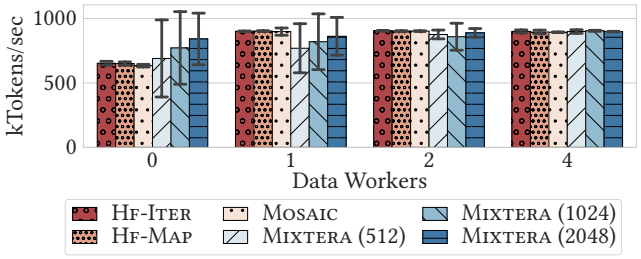


Figure 7: Data loader throughput depending on the number of workers. The number in brackets indicates the chunk size.

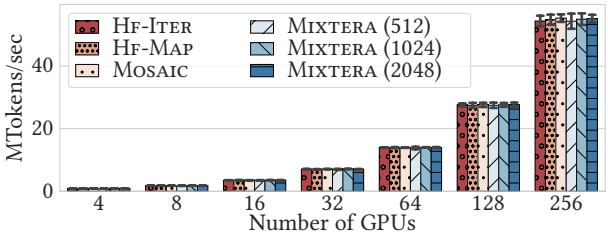


Figure 8: Data loader throughput when increasing the number of data parallel nodes.

increase further. The error bars show the standard deviation of the throughput. Without data workers, MIXTERA has the highest average throughput; with one worker, the other data loaders reach their peak performance.

MIXTERA performs worse than the other loaders with a lower number of workers due to the random access into the files. For every sample, it needs to open the file, seek to the correct position, and load the data, instead of bulk-transferring all the data as the other data loaders can do. This leads to the higher variance in throughput indicated by the error bars, which overall leads to a lower average. However, with a higher number of workers, the variance in data access gets hidden. When using 4+ workers, MIXTERA performs almost identical to the other data loaders. Additionally, increasing the chunk size also helps, and for the 0 worker case, MIXTERA with a chunk size of 1024 or 2048 even has a higher average throughput.

This benchmark setup is quite extreme, as we train a very small 162M model on an extremely fast GPU. Nevertheless, just by adding some workers, which comes at no overhead beyond some CPU resources, MIXTERA reaches competitive throughput despite its more complicated model.

Scaling out. We investigate how the data loaders scale for larger training jobs with higher data parallelism. We scale up to 64 GH200 nodes with a total of 256 data parallel GPUs. We find that using a maximum number of 16 replication GPUs works best. For 4, 8, and 16 GPUs, we use half the GPUs as replication, and shard across the rest. The results with 8 data workers can be found in Figure 8. All data loaders scale perfectly linear. For larger number of GPUs, the throughput variance increases a bit for all systems. We attribute this to the random assignment of nodes in the cluster by the slurm scheduler across the 3 repetitions. We do not test pipeline or tensor parallelism as (i) this is not necessary for the 162M model (and a larger model would only stress the data loaders less), (ii) the data loaders besides MIXTERA do not easily support 3D parallelism, (iii) increasing data parallelism increases the load on the system more.

File formats. All previous benchmarks use uncompressed `jsonl` data. We test compressed `jsonl` (`jsonl.zst`), `parquet`, and the `webdatasets` format in the data loaders that support them. Notably, only MIXTERA supports all of these formats. We find that the underlying file format does not impact the training throughput and therefore omit a plot. This observation holds across different numbers of data workers, where we also observe minimal performance variation among the data loaders.

6.3.1 Query execution. MIXTERA executes the query in DuckDB and generates the `ChunkerIndex` before we can start training. On our Pile dataset with 241 M samples, DuckDB takes 29 s, and preparing the index takes 18 s due to our C++ implementation. Optionally, we can also persist an initial state checkpoint of the query, which due to serialization of nested Python dictionaries takes 13 min. This is only necessary if checkpointing should be supported. Future checkpoints can then be stored in milliseconds. While the streaming HF-ITER data loader can basically start streaming data immediately, the HF-MAP data loader, the default in nanotron, loads and tokenizes the data first, taking 2 h 51 min for this dataset. However, what needs to be considered is that for streaming data loaders such as HF-ITER or MOSAIC, in many scenarios, users would also need to run more offline preprocessing, e.g., to perform static filtering, or reshuffling the data if we want to mix on a different property. MIXTERA avoids this offline preprocessing completely, and, besides an optional state checkpoint, starts streaming data in under a minute.

7 CONCLUSION AND FUTURE WORK

Managing training data at scale requires data management support. We present MIXTERA, a training framework-agnostic data plane for foundation model training. Using its chunking mechanism, it supports dynamic mixtures across arbitrary properties. We demonstrate MIXTERA's performance and scalability, as well as the importance of mixtures on both LLMs and VLMs. For future work, it is interesting to extend the analysis to the cloud, i.e., using a service like S3 as the underlying storage layer. MIXTERA also lays the foundation to implement lineage tracking on which model was trained on which data, as it is a centralized access point. From a ML perspective, the impact of data composition and quality on model performance is still under-explored, and MIXTERA provides a foundation to conduct future research on dynamic and static mixing algorithms.

ACKNOWLEDGMENTS

We thank Dan Graur, Viktor Gsteiger, and Beste Güney for their contributions to MIXTERA's codebase. We thank Antoni-Joan Solergibert i Llaquet, Imanol Schlag, Steven Hand, Martin Jaggi, Antoine Bosselut, Alexander Hägele, Loubna Ben Allal, Quentin de Laroussilhe, Paul Barham, Yiding Jiang, Theodoros Rekatsinas, and Foteini Strati for helpful discussions. This work was supported as part of the Swiss AI Initiative by a grant from the Swiss National Supercomputing Centre (CSCS) under project ID a09 on Alps. Maximilian Böther is supported by the Swiss National Science Foundation (project number 200021_204620).

REFERENCES

[1] 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, 12 (2015). doi:10.14778/2824032.2824076

[2] Ibrahim M. Alabdulmohsin, Xiaohua Zhai, Alexander Kolesnikov, and Lucas Beyer. 2023. Getting ViT in Shape: Scaling Laws for Compute-Optimal Model Design. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*.

[3] Alon Albalak, Liangming Pan, Colin Raffel, and William Yang Wang. 2023. Efficient Online Data Mixing For Language Model Pre-Training. *arXiv Preprint* (2023). doi:10.48550/arXiv.2312.02406

[4] Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Gabriel Martín Blázquez, Guilherme Penedo, Lewis Tunstall, Andrés Marafioti, Hynek Kydlíček, Agustín Piqueres Lajarín, Vaibhav Srivastav, Joshua Lochner, Caleb Fahlgren, Xuan-Son Nguyen, Clémentine Fourrier, Ben Burtenshaw, Hugo Larcher, Haojun Zhao, Cyril Zakka, Mathieu Morlon, Colin Raffel, Leandro von Werra, and Thomas Wolf. 2025. SmolLM2: When Smol Goes Big – Data-Centric Training of a Small Language Model. *arXiv preprint* (2025). doi:10.48550/arXiv.2502.02737

[5] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[6] Tal Ben-Nun and Torsten Hoefler. 2019. Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis. *Comput. Surveys* 52, 4 (2019), 1–43. doi:10.1145/3320060

[7] Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. 2020. PIQA: Reasoning about Physical Commonsense in Natural Language. *Proceedings of the Conference on Artificial Intelligence (AAAI)*. doi:10.1609/aaai.v34i05.6239

[8] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Dora Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu Ma, Ali Malik, Christopher D. Manning, Suvir Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Ben Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, Julian Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Rob Reich, Hongyu Ren, Frieda Rong, Yusuf Roohani, Camilo Ruiz, Jack Ryan, Christopher Ré, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishnan Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga, Jiaxuan You, Matei Zaharia, Michael Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. 2022. On the Opportunities and Risks of Foundation Models. In *arXiv preprint*. doi:10.48550/arXiv.2108.07258

[9] 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. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*.

[10] Daoyuan Chen, Yilun Huang, Zhijian Ma, Hesen Chen, Xuchen Pan, Ce Ge, Dawei Gao, Yuexiang Xie, Zhaoyang Liu, Jinyang Gao, Yaliang Li, Bolin Ding, and Jingren Zhou. 2024. Data-Juicer: A One-Stop Data Processing System for Large Language Models. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. doi:10.1145/3626246.3653385

[11] Mayee F. Chen, Michael Y. Hu, Nicholas Lourie, Kyunghyun Cho, and Christopher Ré. 2024. Aioli: A Unified Optimization Framework for Language Model Data Mixing. In *Proceedings of the International Conference on Learning Representations (ICLR)*. doi:10.48550/arXiv.2411.05735

[12] Mayee F. Chen, Nicholas Roberts, Kush Bhatia, Jue Wang, Ce Zhang, Frederic Sala, and Christopher Ré. 2023. Skill-it! A Data-Driven Skills Framework for Understanding and Training Language Models. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*. doi:10.48550/arXiv.2307.14430

[13] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. 2018. Think you have Solved Question Answering? Try ARC, the AI2 Reasoning Challenge. *arXiv preprint* (2018). doi:10.48550/arXiv.1803.05457

[14] Competition and Markets Authority. 2013. *AI Foundation Models: Initial Report.* Technical Report. UK Government Agency.

[15] A. Feder Cooper, Jonathan Frankle, and Christopher De Sa. 2022. Non-Determinism and the Lawlessness of Machine Learning Code. In *Proceedings of the Symposium on Computer Science and Law (CSLAW)*. doi:10.1145/3511265.3550446

[16] ETH AI Center & EPFL AI Center. 2025. Swiss AI Initiative. https://www.swiss-ai.org/.

[17] Chaoyou Fu, Peixian Chen, Yunhang Shen, Yulei Qin, Mengdan Zhang, Xu Lin, Jinrui Yang, Xiawu Zheng, Ke Li, Xing Sun, Yunsheng Wu, and Rongrong Ji. 2024. MME: A Comprehensive Evaluation Benchmark for Multimodal Large Language Models. *arXiv preprint* (2024). doi:10.48550/ARXIV.2306.13394

[18] Luigi Fusco, Mikhail Khalilov, Marcin Chrapek, Giridhar Chukkapalli, Thomas C. Schulthess, and Torsten Hoefler. 2024. Understanding Data Movement in Tightly Coupled Heterogeneous Systems: A Case Study with the Grace Hopper Superchip. *arXiv preprint* (2024). doi:10.48550/ARXIV.2408.11556

[19] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. 2020. The Pile: An 800GB Dataset of Diverse Text for Language Modeling. *arXiv preprint* (2020). doi:10.48550/arXiv.2101.00027

[20] Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. 2023. A framework for few-shot language model evaluation. doi:10.5281/zenodo.10256836

[21] Torsten Hoefler, Tommaso Bonoto, Daniele De Sensi, Salvatore Di Girolamo, Shigang Li, Marco Heddes, Deepak Goel, Miguel Castro, and Steve Scott. 2024. HammingMesh: A Network Topology for Large-Scale Deep Learning. *Commun. ACM* 67, 12 (2024), 97–105. doi:10.1145/3623490

[22] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*.

[23] Drew A. Hudson and Christopher D. Manning. 2019. GQA: A New Dataset for Real-World Visual Reasoning and Compositional Question Answering. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*. doi:10.1109/cvpr.2019.00686

[24] HuggingFace. 2025. *Nanotron: Pretraining models made easy.* https://github.com/huggingface/nanotron

[25] Junlong Jia, Ying Hu, Xi Weng, Yiming Shi, Miao Li, Xingjian Zhang, Baichuan Zhou, Ziyu Liu, Jie Luo, Lei Huang, and Ji Wu. 2024. TinyLLaVA Factory: A Modularized Codebase for Small-scale Large Multimodal Models. *arXiv preprint* (2024). doi:10.48550/arXiv.2405.11788

[26] Yiding Jiang, Allan Zhou, Zhili Feng, Sadhika Malladi, and J. Zico Kolter. 2024. Adaptive Data Optimization: Dynamic Sample Selection with Scaling Laws. In *Proceedings of the International Conference on Learning Representations (ICLR)*. doi:10.48550/arXiv.2410.11820

[27] Siddharth Karamcheti, Laurel Orr, Jason Bolton, Tianyi Zhang, Karan Goel, Avanika Narayan, Rishi Bommasani, Deepak Narayanan, Tatsunori Hashimoto, Dan Jurafsky, Christopher D. Manning, Christopher Potts, Christopher Ré, and Percy Liang. 2021. Mistral - A Journey towards Reproducible Language Model Training.

[28] Ranjay Krishna, Yuke Zhu, Oliver Groth, Justin Johnson, Kenji Hata, Joshua Kravitz, Stephanie Chen, Yannis Kalantidis, Li-Jia Li, David A. Shamma, Michael S. Bernstein, and Li Fei-Fei. 2017. Visual Genome: Connecting Language and Vision Using Crowdsourced Dense Image Annotations. *International Journal of Computer Vision* 123, 1 (2017). doi:10.1007/s11263-016-0981-7

[29] Michael Kuchnik, Ana Klimovic, Jiri Simsa, Virginia Smith, and George Amvrosiadis. 2022. Plumber: Diagnosing and Removing Performance Bottlenecks in Machine Learning Data Pipelines. In *Proceedings of the Conference on Machine Learning and Systems (MLSys)*.

[30] Yifan Li, Yifan Du, Kun Zhou, Jinpeng Wang, Xin Zhao, and Ji-Rong Wen. 2023. Evaluating Object Hallucination in Large Vision-Language Models. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*. doi:10.18653/v1/2023.emnlp-main.20

[31] Zeman Li, Yuan Deng, Peilin Zhong, Meisam Razaviyayn, and Vahab Mirrokni. 2025. PiKE: Adaptive Data Mixing for Multi-Task Learning Under Low Gradient Conflicts. *arXiv preprint* (2025). doi:10.48550/arXiv.2502.06244

[32] Wanchao Liang, Tianyu Liu, Less Wright, Will Constable, Andrew Gu, Chien-Chin Huang, Iris Zhang, Wei Feng, Howard Huang, Junjie Wang, Sanket Purandare, Gokul Nadathur, and Stratos Idreos. 2024. TorchTitan: One-stop PyTorch native solution for production ready LLM pre-training. *arXiv preprint* (2024).

doi:10.48550/arXiv.2410.06511

[33] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. 2014. Microsoft COCO: Common Objects in Context. In *Proceedings of the European Conference on Computer Vision (ECCV)*. doi:10.1007/978-3-319-10602-1_48

[34] Haotian Liu, Chunyuan Li, Yuheng Li, and Yong Jae Lee. 2024. Improved Baselines with Visual Instruction Tuning. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*. doi:10.1109/cvpr52733.2024.02484

[35] Hanmeng Liu, Jian Liu, Leyang Cui, Zhiyang Teng, Nan Duan, Ming Zhou, and Yue Zhang. 2023. LogiQA 2.0—An Improved Dataset for Logical Reasoning in Natural Language Understanding. *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 31 (2023). doi:10.1109/taslp.2023.3293046

[36] Shayne Longpre, Gregory Yauney, Emily Reif, Katherine Lee, Adam Roberts, Barret Zoph, Denny Zhou, Jason Wei, Kevin Robinson, David Mimno, and Daphne Ippolito. 2024. A Pretrainer's Guide to Training Data: Measuring the Effects of Data Age, Domain Coverage, Quality, & Toxicity. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*. doi:10.18653/v1/2024.naacl-long.179

[37] Pan Lu, Swaroop Mishra, Tony Xia, Liang Qiu, Kai-Wei Chang, Song-Chun Zhu, Oyvind Tafjord, Peter Clark, and Ashwin Kalyan. 2022. Learn to Explain: Multimodal Reasoning via Thought Chains for Science Question Answering. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*.

[38] Meta. 2024. The Llama 3 Herd of Models. *arXiv preprint* (2024). doi:10.48550/arXiv.2407.21783

[39] Meta. 2024. Llama 3.3 Model Card. https://github.com/meta-llama/llama-models/blob/main/models/llama3_3/MODEL_CARD.md. Accessed: 2024-12-18.

[40] Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. 2018. Can a Suit of Armor Conduct Electricity? A New Dataset for Open Book Question Answering. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*. doi:10.18653/v1/d18-1260

[41] Anand Mishra, Shashank Shekhar, Ajeet Kumar Singh, and Anirban Chakraborty. 2019. OCR-VQA: Visual Question Answering by Reading Text in Images. In *Proceedings of the International Conference on Document Analysis and Recognition (ICDAR)*. doi:10.1109/icdar.2019.00156

[42] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. 2021. Analyzing and mitigating data stalls in DNN training. *Proceedings of the VLDB Endowment* 14, 5 (2021). doi:10.14778/3446095.3446100

[43] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. 2021. tf.data: a machine learning data processing framework. *Proceedings of the VLDB Endowment* 14, 12 (2021). doi:10.14778/3476311.3476374

[44] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. doi:10.1145/3341301.3359646

[45] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. doi:10.1145/3458817.3476209

[46] OpenAI. 2024. GPT-4 Technical Report. In *arXiv preprint*. doi:10.48550/arXiv.2303.08774

[47] Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Ngoc Quan Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernandez. 2016. The LAMBADA dataset: Word prediction requiring a broad discourse context. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*. doi:10.18653/v1/p16-1144

[48] Guilherme Penedo, Hynek Kydlíček, Loubna Ben allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. 2024. The FineWeb Datasets: Decanting the Web for the Finest Text Data at Scale. *arXiv preprint* (2024). doi:10.48550/arXiv.2406.17557

[49] Guilherme Penedo, Hynek Kydlíček, Alessandro Cappelli, Mario Sasko, and Thomas Wolf. 2024. *DataTrove: large scale data processing*. https://github.com/huggingface/datatrove

[50] Shangshu Qian, Hung Viet Pham, Thibaud Lutellier, Zeou Hu, Jungwon Kim, Lin Tan, Yaoliang Yu, Jiahao Chen, and Sameena Shah. 2021. Are My Deep Learning Systems Fair? An Empirical Study of Fixed-Seed Training. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*.

[51] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. doi:10.1145/3299869.3320212

[52] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. In *Self-hosted preprint*.

[53] Aquia Richburg and Marine Carpuat. 2024. How Multilingual Are Large Language Models Fine-Tuned for Translation? *arXiv preprint* (2024). doi:10.48550/arXiv.2405.20512

[54] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2021. WinoGrande: An Adversarial Winograd Schema Challenge at Scale. *Commun. ACM* 64, 9 (2021). doi:10.1145/3474381

[55] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake A. Hechtman. 2018. Mesh-TensorFlow: Deep Learning for Supercomputers. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*.

[56] Zhiqiang Shen, Tianhua Tao, Liqun Ma, Willie Neiswanger, Zhengzhong Liu, Hongyi Wang, Bowen Tan, Joel Hestness, Natalia Vassilieva, Daria Soboleva, and Eric Xing. 2024. SlimPajama-DC: Understanding Data Combinations for LLM Training. *arXiv preprint* (2024). 10.48550/arXiv.2309.10818

[57] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint* (2019). doi:10.48550/arXiv.1909.08053

[58] Amanpreet Singh, Vivek Natarajan, Meet Shah, Yu Jiang, Xinlei Chen, Dhruv Batra, Devi Parikh, and Marcus Rohrbach. 2019. Towards VQA Models That Can Read. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*. doi:10.1109/cvpr.2019.00851

[59] Daria Soboleva, Faisal Al-Khateeb, Robert Myers, Jacob R Steeves, Joel Hestness, and Nolan Dey. 2023. SlimPajama: A 627B token cleaned and deduplicated version of RedPajama. https://huggingface.co/datasets/cerebras/SlimPajama-627B

[60] Luca Soldaini, Rodney Kinney, Akshita Bhagia, Dustin Schwenk, David Atkinson, Russell Authur, Ben Bogin, Khyathi Chandu, Jennifer Dumas, Yanai Elazar, Valentin Hofmann, Ananya Harsh Jha, Sachin Kumar, Li Lucy, Xinxi Lyu, Nathan Lambert, Ian Magnusson, Jacob Morrison, Niklas Muennighoff, Aakanksha Naik, Crystal Nam, Matthew E. Peters, Abhilasha Ravichander, Kyle Richardson, Zejiang Shen, Emma Strubell, Nishant Subramani, Oyvind Tafjord, Pete Walsh, Luke Zettlemoyer, Noah A. Smith, Hannaneh Hajishirzi, Iz Beltagy, Dirk Groeneveld, Jesse Dodge, and Kyle Lo. 2024. Dolma: An Open Corpus of Three Trillion Tokens for Language Model Pretraining Research. *arXiv preprint* (2024). doi:10.48550/arXiv.2402.00159

[61] The LlaVA Team. 2023. *LLaVA Data Documentation*. https://github.com/haotian-liu/LLaVA/blob/main/docs/Data.md

[62] The LlaVA Team. 2024. *TinyLLaVA: Model Zoo*. https://github.com/TinyLLaVA/TinyLLaVA_Factory?tab=readme-ov-file#model-zoo

[63] The Mosaic ML Team. 2022. *streaming: Fast, accurate streaming of training data from cloud storage*. https://github.com/mosaicml/streaming/

[64] The TensorFlow Team. 2025. *Tensorflow: Determinism*. https://www.tensorflow.org/api_docs/python/tf/config/experimental/enable_op_determinism

[65] The TinyLLaVA Factory Team. 2024. *TinyLLaVA Factory: Prepare Datasets*. https://tinyllava-factory.readthedocs.io/en/latest/Prepare%20Datasets.html

[66] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*.

[67] Zige Wang, Wanjun Zhong, Yufei Wang, Qi Zhu, Fei Mi, Baojun Wang, Lifeng Shang, Xin Jiang, and Qun Liu. 2023. Data Management For Large Language Models: A Survey. *arXiv preprint* (2023). doi:10.48550/ARXIV.2312.01700

[68] Maurice Weber, Daniel Y Fu, Quentin Gregory Anthony, Yonatan Oren, Shane Adams, Anton Alexandrov, Xiaozhong Lyu, Huu Nguyen, Xiaozhe Yao, Virginia Adams, Ben Athiwaratkun, Rahul Chalamala, Kezhen Chen, Max Ryabinin, Tri Dao, Percy Liang, Christopher Re, Irina Rish, and Ce Zhang. 2024. RedPajama: an Open Dataset for Training Large Language Models. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*.

[69] Sang Michael Xie, Hieu Pham, Xuanyi Dong, Nan Du, Hanxiao Liu, Yifeng Lu, Percy S Liang, Quoc V Le, Tengyu Ma, and Adams Wei Yu. 2024. Doremi: Optimizing data mixtures speeds up language model pretraining. *Advances in Neural Information Processing Systems* 36 (2024).

[70] Canwen Xu, Corby Rosset, Ethan C. Chau, Luciano Del Corro, Shweti Mahajan, Julian McAuley, Jennifer Neville, Ahmed Hassan Awadallah, and Nikhil Rao. 2024. Automatic Pair Construction for Contrastive Post-training. *arXiv preprint* (2024). 10.48550/arXiv.2310.02263

[71] Haoran Xu, Young Jin Kim, Amr Sharaf, and Hany Hassan Awadalla. 2024. A Paradigm Shift in Machine Translation: Boosting Translation Performance of Large Language Models. In *Proceedings of the ML Evaluation Standards Workshop at ICLR*. doi:10.48550/arXiv.2309.1167

[72] Jiasheng Ye, Peiju Liu, Tianxiang Sun, Yunhua Zhou, Jun Zhan, and Xipeng Qiu. 2024. Data Mixing Laws: Optimizing Data Mixtures by Predicting Language Modeling Performance. *arXiv preprint* (2024). doi:10.48550/arXiv.2403.16952

[73] Xiang Yue, Yuansheng Ni, Tianyu Zheng, Kai Zhang, Ruoqi Liu, Ge Zhang, Samuel Stevens, Dongfu Jiang, Weiming Ren, Yuxuan Sun, Cong Wei, Botao Yu, Ruibin Yuan, Renliang Sun, Ming Yin, Boyuan Zheng, Zhenzhu Yang, Yibo Liu, Wenhao Huang, Huan Sun, Yu Su, and Wenhu Chen. 2024. MMMU: A Massive Multi-Discipline Multimodal Understanding and Reasoning Benchmark for Expert AGI. In *Proceedings of the Conference on Computer Vision and Pattern*

Recognition (CVPR). doi:10.1109/cvpr52733.2024.00913

[74] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65. doi:10.1145/2934664

[75] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. HellaSwag: Can a Machine Really Finish Your Sentence?. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*. doi:10.18653/v1/p19-1472

[76] Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. 2024. TinyLlama: An Open-Source Small Language Model. *arXiv preprint* (2024). doi:10.48550/arXiv.2401.02385

[77] 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. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*. doi:10.1145/3470496.3533044

[78] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proceedings of the VLDB Endowment* 16, 12 (2023). doi:10.14778/3611540.3611569

[79] Donglin Zhuang, Xingyao Zhang, Shuaiwen Song, and Sara Hooker. 2022. Randomness in Neural Network Training: Characterizing the Impact of Tooling. In *Proceedings of the Conference on Machine Learning and Systems (MLSys)*.