Decluttering the data mess in LLM training

Maximilian Böther
ETH Zurich
Switzerland
mboether@ethz.ch

Dan Graur
ETH Zurich
Switzerland
dgraur@ethz.ch

Xiaozhe Yao
ETH Zurich
Switzerland
xiaozhe.yao@ethz.ch

Ana Klimovic
ETH Zurich
Switzerland
aklimovic@ethz.ch

1 Introduction and motivation

Training large language models (LLMs) 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 LLMs train over trillions of tokens from aggregated data collections such as RedPajama [7], Dolma [16], or FineWeb [12]. For example, Meta's `Llama-3-405B` model is trained on a corpus of 15 trillion tokens [9].

As data collections grow to include data with different characteristics that come from different sources, managing the data and selecting which samples to use for training becomes time-consuming, tedious, and prone to errors. When training an LLM, developers first collect training data from various sources, such as Wikipedia, Common Crawl, or ArXiv. They then clean the data, which typically involves deduplicating, filtering, and applying classifiers to the data samples (e.g., to obtain a toxicity score for each sample). Finally, for each training job, developers then need to select and mix data samples, e.g., train on 50 % data from Wikipedia and 50 % from movie subtitles. 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.

Selecting the right proportions of data with particular characteristics (e.g., language, topic, source) for training is critical for model performance [5, 18, 21]. Algorithms like DoReMi [18], SlimPajama-DC [15], or the data mixing laws [21] find the best mixture by trying out combinations on a small proxy model. Xie et al. [18] show that by optimizing data proportions from different sources in The Pile [8] with DoReMi, they reach the same accuracy as using the default data proportions while taking only 38.4 % of the training steps. When training for the same number of steps as the baseline, one-shot accuracy increases by 6.5 percent points.

While efficiently managing and mixing data collections is critical, we observe several challenges due to the lack of system support. We lay out these challenges and then present Mixtera, a work-in-progress system supporting data management in the context of LLM training.

Challenge 1: High engineering effort for data preparation. The current approach for preparing data mixtures involves many manual offline steps with general-purpose data processing and scripting frameworks. Developers typically leverage general-purpose data processing frameworks like Spark [22] or Beam [2] to clean data offline. Afterwards, they spend a lot of time writing error-prone ad-hoc scripts to mix the cleaned data for each training job. Each training job with a new mixture requires a new copy of the dataset and a new mixing script. Since mixtures are prepared offline, training a model with a new mixture requires waiting for the data mixing pipeline to finish executing on the whole source datasets first. This makes it difficult to get a quick sense of how a data mixture will impact model training when exploring different mixing policies.

Challenge 2: Lack of data management system. Current database management systems are not natively built for LLM data [17]. Using a DBMS for LLM 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. Hence, most LLM training data is 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 as `jsonl` (e.g., RedPajama [7]) or `parquet` (e.g., FineWeb [12]) files in a shared filesystem. By the nature of these formats, each sample is stored as a row in the file and the sample-level metadata is stored as a field in the row. When the dataset is extensively labeled with classifiers, the metadata can take up to 30 % of the storage space [6], and is duplicated for each copy of the data generated in the mixing process. This leads to high *storage overhead*.

Furthermore, filesystems do not provide an efficient interfaces to query data. As a result, if developers want to find all data that satisfies a condition, they have to scan the entire dataset, which can lead to *performance bottlenecks*.

The lack of data management system also makes it hard to guarantee *consistency*. For example, when a data collection is re-processed with a new personal identifiable information (PII) removal pipeline that classifies more data as PII, it is important but difficult to guarantee that old data mixtures which may contain PII data are removed. Lineage tracking, i.e., tracking which models are trained with which data, is also not supported by the filesystem.

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 being developed. There are many open questions, such as with which granularity to guarantee data mixtures. DoReMi guarantees the mixture within a batch [18], but a larger window may be sufficient.

Furthermore, it might be useful to specify hierarchical mixtures across arbitrary properties, e.g., specify a proportion between Wikipedia and movie subtitles, *and* between English and German. The order in which data is fed to a model also impacts accuracy. Curriculum learning pre-defines a data order. Xu et al. [19] order samples from *easy* to *hard* to improve alignment. Multilingual models are often first trained on English data, and then data from other languages is included in the mixture [14, 20]. 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. Skill-it is a framework to order "skills" based on model feedback [5]. Albalak et al. [3] use a multi-armed bandit strategy to dynamically change the mixture. However, unfortunately even for researchers who are familiar with the latest techniques, implementing them in a training pipeline is a painful, tedious, and error-prone task. This hinders the adoption of new methods and makes researching, reproducing, and comparing different strategies difficult.

2 Mixtera: A lightweight LLM training data lake

We are currently building Mixtera, a lightweight data lake for distributed LLM training.

2.1 Design Goals

Based on the aforementioned challenges, we identify design goals for an LLM training data management system:

Goal G1: The system should implement a *centralized* data layer that users can conveniently and *declaratively* query based on metadata. During training, the data should be *streamed on-the-fly* adhering to the mixture.

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

Goal G3: The system should support *adjusting the mixture dynamically* during the training by offering the user callbacks to adjust the mixture.

Goal G4: While being user-friendly and flexible, the system needs to ensure *high-throughput*, even in distributed training, despite streaming data over the network.

2.2 Mixtera design

Inspired by the Lakehouse architecture [4], Mixtera is as a lightweight, read-only layer on top of training data collections (see Figure 1). For each sample, Mixtera stores user-defined properties (e.g., language, topic, or source dataset). It does not actively reorganize or modify the files on disk. It can be easily integrated into existing setups, such as collections of `jsonl` files, and allows for declarative queries (G1). It returns a stream of chunks, which contain pointers to samples conforming to the mixture specified in the query.

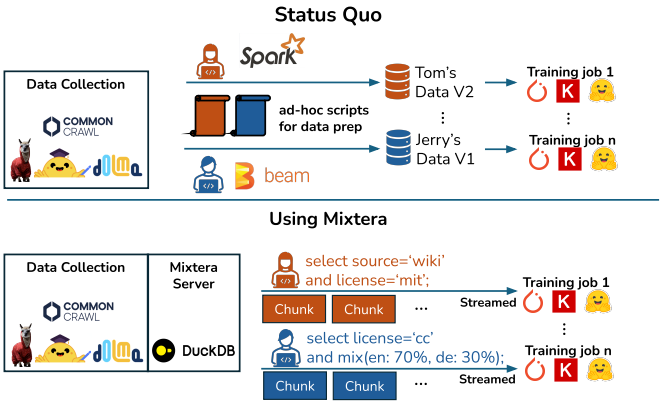

Figure 1. Mixtera System Overview

```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 2. An example query using Mixtera.

Due to Mixtera's training framework agnostic interface, the samples can be directly fed into any training pipeline.

Users interact with Mixtera in two stages. First, they register datasets in the system and add properties of the samples to populate the database. The properties can be obtained from the source files themselves (e.g., which sub-dataset the sample stems from) or by running classifiers (e.g., to identify the language or toxicity). Mixtera has a programmatical interface for this step, and we plan to implement a CLI.

In the second phase, users submit SQL-like queries and run trainings. In Figure 2, we show a minimal example for a query which statically filters out only creative commons data, and then mixes HTML and JavaScript data in a 70:30 ratio. Mixtera abstractions such as `MixteraTorchDataset` take care of submitting the query and obtaining the samples without needing to worry about correctness in distributed training (G2, G4). The user only needs to provide the ID of the node and its data parallel group, which can be obtained from the training framework like nanotron.

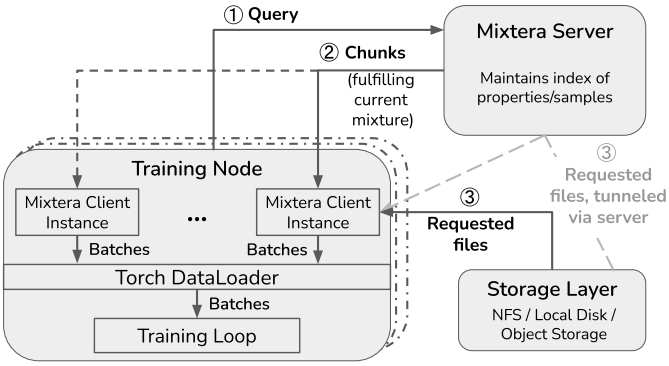

Figure 3. Data flow in Mixtera.

Data flow and overview. Mixtera comes as a Python package. It is independent of the training framework as it provides a standard PyTorch [11] `IterableDataset` abstraction. In distributed training, the Mixtera server runs on a node and each training node runs Mixtera client instances. Mixtera can also be used as a Python module without networking in case of single-GPU training.

Figure 3 shows the basic dataflow we envision for the client-server setup. The client (e.g., a training process) submits a query to Mixtera. The query is executed at the server in two phases. In the first phase, Mixtera applies static filters from the query to get a query result with all matching samples. In the second phase, the server starts distributing *chunks* of that query result to the client(s). A chunk is a collection of pointers to samples in files. These pointers tell the receiving client which samples in the file to consume. The server generates chunks according to the current mixture, i.e., it iteratively cuts off pieces of the overall query result such that the samples in the chunk follow the intended mixture. The server ensures that the chunks are distributed correctly, i.e., tensor- and pipeline parallel stages receive the same input data.

Having received a chunk, the clients then fetch the data based on the pointers in the chunk. The files that the chunks point to can either be local, or on a network drive, or potentially even cloud storage. The data can also be tunneled via the server, e.g., if it is only available on a server-local disk.

Why store metadata only? We opt to implement centralization (G1) on a metadata level: the index structure stores which samples in which files have which properties. We do not store the samples themselves within Mixtera. Distributing chunks containing pointers instead of actual sample payloads has several advantages. First, we want to be flexible and not force users to actually store the data in one central location (G2), although we recommend it for consistency. This approach allows users to store data in their location of choice (e.g., an object store, an NFS, or on each machine) and does not require the Mixtera server to ingest the data. Second, the pointer-model reduces the network load in case the files are on a local disk at each node. Third, we do not force our own file structure on the users and allow for easy adoption on existing data collections (G2). A disadvantage is that, since Mixtera is a read-only layer, we cannot optimize the storage layout of the original data.

2.3 Mixtera implementation

We briefly discuss how we implement different parts of Mixtera's current prototype.

Underlying database. We initially started out building Mixtera with custom Python indices. However, they turned out to be too slow for large datasets. Mixtera now uses DuckDB [13] as an underlying database. DuckDB runs in-process and can be seemlessly embedded into a Python application. It satisfies goal G2, as it does not have any setup overhead, and to users of Mixtera, it is completely transparent. At the same time, DuckDB allows to execute queries very fast. Mixtera indexes every sample in the dataset as a row in a table, assigning it a unique sample ID, and storing user-defined properties, such as language or topic.

Query execution and chunk generation. In the first step of query execution (static filtering), Mixtera generates a selection SQL common table expression (CTE) based on the filters. Then, it generates a query using that CTE and windowing functions to obtain a list of intervals of samples. For example, if samples x to y all share the same properties, DuckDB will return an interval $\langle x, y \rangle$ together with its properties. We generate a data structure from this result called a `ChunkerIndex`, from which we can can efficiently slice off chunks according to the mixture. Since we implement the chunking as an iterator, the mixture can be updated on-the-fly.

Server and chunk distribution. The server distributes chunks via TCP and asyncio. It caches chunks until all nodes in the same data parallel group have consumed it. This is important, since all nodes within a data parallel group need to read exactly the same data in exactly the same order.

Client-side reading. Mixtera abstracts different file formats (e.g., `jsonl`, `parquet` or `Crossaint` [1]) and file systems (i.e., S3 or local filesystem). We implement multi-threaded reading from files, while having to ensure mixture-correct reading (batches might be smaller than chunks) and, importantly, reproducibility across nodes, as each node needs to yield samples in exactly the same order.

2.4 Outlook

We are currently benchmarking Mixtera's training throughput and comparing to other implementations, such as huggingface datasets. We are testing our design's scalability to see whether Mixtera's client-server architecture is sufficient to avoid data stalls. We plan to implement a callback-based feedback loop for dynamically adjusting the mixture, such as in Skill-it [5]. In the long term, we also aim to support multi-modal data, which brings additional challenges, such as image decoding and augmentation [10].

References

[1] Mubashara Akhtar, Omar Benjelloun, Costanza Conforti, Pieter Gijsbers, Joan Giner-Miguelez, Nitisha Jain, Michael Kuchnik, Quentin Lhoest, Pierre Marcenac, Manil Maskey, Peter Mattson, Luis Oala, Pierre Ruyssen, Rajat Shinde, Elena Simperl, Goeffry Thomas, Slava Tykhonov, Joaquin Vanschoren, Jos van der Velde, Steffen Vogler, and Carole-Jean Wu. 2024. Croissant: A Metadata Format for ML-Ready Datasets. In *Proceedings of the Workshop on Data Management for End-to-End Machine Learning (DEEM)*. ACM. https://doi.org/10.1145/3650203.3663326

[2] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment* 8, 12 (2015). https://doi.org/10.14778/2824032.2824076

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

[4] 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)*.

[5] 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. *arXiv preprint* (2023). https://doi.org/10.48550/arXiv.2307.14430

[6] Together Computer. 2023. File taken from RedPajama-V2. https://huggingface.co/datasets/togethercomputer/RedPajama-Data-V2/resolve/refs%2Fconvert%2Fparquet/default/partial-train/0000.parquet

[7] Together Computer. 2023. *RedPajama: an Open Dataset for Training Large Language Models*. https://github.com/togethercomputer/RedPajama-Data

[8] 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). https://doi.org/10.48550/arXiv.2101.00027

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

[10] 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). https://doi.org/10.14778/3476311.3476374

[11] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[12] 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). https://doi.org/10.48550/arXiv.2406.17557

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

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

[15] 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

[16] 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). https://doi.org/10.48550/arXiv.2402.00159

[17] 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). https://doi.org/10.48550/ARXIV.2312.01700

[18] 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).

[19] 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

[20] 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*. https://doi.org/10.48550/arXiv.2309.1167

[21] 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). https://doi.org/10.48550/arXiv.2403.16952

[22] 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. https://doi.org/10.1145/2934664