MLDataForge: Accelerating Large-Scale Dataset Preprocessing and Access for Multimodal Foundation Model Training

Andrea Blasi Núñez, Lukas Paul Achatius Galke, Peter Schneider-Kamp
University of Southern Denmark
{abln,galke,petersk}@imada.sdu.dk

Abstract

Preprocessing large and possibly multimodal datasets remains a key bottleneck in many machine learning workflows, particularly when random access to samples is needed for global shuffling and sorting. Existing approaches, including widely used formats like JSONL and frameworks such as Huggingface Datasets and MosaicML Streaming, typically incur substantial computational, memory, and storage overhead in such settings. Here, we introduce MLDataForge, a Python-based open-source framework designed for scalable dataset preprocessing and access. Our key contributions are: (1) optimized readers for Mosaic Data Shards (MDS) that substantially improve throughput, reduce peak storage usage, and support sample-level compression; (2) JINX (JSON Indexed 'N' eXtended), a novel, index-augmented JSONL-compatible format supporting structured footers and binary sidecar files; and (3) a lazy loading mechanism that defers data loading, decompression, and decoding JINX files until sample fields are accessed. We empirically evaluate MLDataForge and our contributions on a representative 200 GB supervised fine-tuning dataset for vision language models. Our best configuration – zstd-compressed JINX with binary sidecar and lazy loading – yields at least a decimal order-of-magnitude throughput increase compared to the best baselines for iteration, global shuffling, and sorting. These advances enable substantial gains in data preprocessing performance, facilitating more scalable and resource-efficient model training pipelines.

1 Introduction

Handling large-scale datasets is essential for training multi-modal foundation models, especially with the growing importance of multimodal data that combines text, images, audio, and more. However, working with such datasets introduces major performance challenges (Ji et al., 2025). Common operations like shuffling, sorting, or even just iterating through large datasets can become very slow and memory-intensive when using standard tools and formats (Knuth, 1998). This presents a crucial bottleneck in pre-processing and training pipelines and makes reproducibility and scalability harder to achieve.

JSON Lines (JSONL) is likely the most commonly used format for large-scale text datasets due to its simplicity, flexibility, and human readability (Gogula, 2025). However, JSONL is not well-geared for multi-modal content as it requires application-specific encoding of binary content. Further, JSONL inherently lacks support for high-performance random access and, consequently, global shuffling and sorting operations.

Mosaic Data Shards (MDS) was introduced as a solution to these limitations (MosaicML Team, 2022). It offers a binary, shard-based format that is notably faster for streaming, supports efficient shuffling, and is optimized for distributed training. However, while Mosaic's MDS format and streaming library represent a major step forward in handling large-scale datasets, they still fall short of the speed required for the most demanding preprocessing and training scenarios.

Therefore, we introduce MLDataForge, an open-source Python package that provides a framework and a command-line interface featuring improved readers and writers for extant data formats and a novel data format that enables order-of-magnitude throughput increases for operations like iteration, global shuffling, and sorting (see Figure 1). These innovations are described in detail in Section 3.

To quantify this performance improvement, we conducted a series of experiments comparing data handling using the HuggingFace Datasets and MosaicML Streaming frameworks with our approach. To that end, we benchmarked key operations inclu-

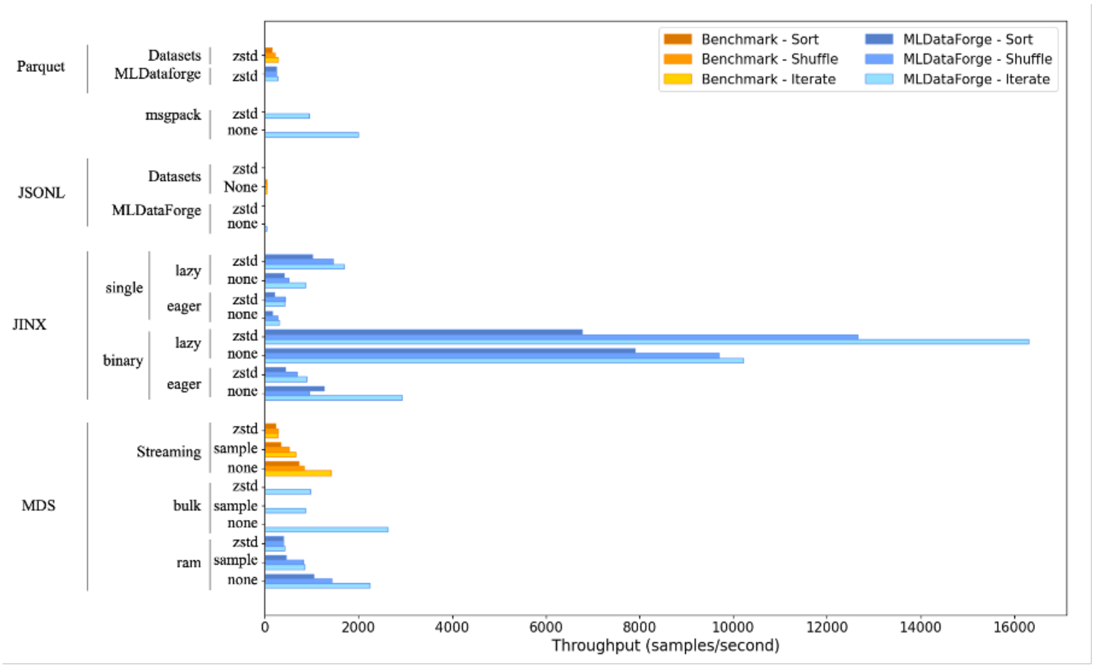

Figure 1: Throughput (samples per second, **higher is better**) for each operation (sort, shuffle, iterate) per file type, data reader, and compression method. Orange bars represent the benchmark performance (our baseline), while blue bars show the performance achieved using MLDataForge. Within each color, shades indicate the type of operation: dark for sorting, medium for shuffling, and light for iterating. For example, dark orange corresponds to benchmark performance during sorting, while dark blue represents MLDataForge performance during the same operation. Our MLDataForge framework, along with our proposed JINX format, binary sidecar, and zstd compression, yields the highest throughput, by an order of magnitude, in all operations.

diong iteration, global shuffling, and sorting given an arbitrary key function. The evaluation was conducted across five data formats: MDS, JSONL, Parquet, MSGPACK, and our custom format, JSON Index N Extended (JINX; see Section 3). While MLDataForge supports many compression types for each of the data formats, for our experiments we restricted ourselves to at most three compression types, if applicable: no compression, Zstandard (zstd) file-level compression, and zstd sample-level compression (see also Section 3).

The immense performance gains came with lower computational costs and resource usage, making it more feasible to work with large-scale multimodal datasets. Ultimately, these advances help accelerate the development and deployment of state-of-the-art machine learning models.

2 Related Work

Some of the best-known frameworks for handling large-scale ML datasets are MDS, Parquet, and HuggingFace Datasets.

MDS is a dataset format developed by MosaicML (now a part of Databricks) specifically designed for handling large-scale multimodal datasets (MosaicML Team, 2022). The format stores data in binary sharded files with a header index. It uses an accompanying JSON file to manage metadata and access. Its strongest features is its streaming-oriented design, which enables efficient iteration even when data is distributed or stored remotely. However, global shuffling, requires either larger buffers or reading the entire dataset, which can increase memory and I/O costs. Sorting is not natively supported and would generally involve loading metadata, such as labels or timestamps, into memory first, which makes sorting slower and more memory-intensive compared to other tools.

Apache Parquet (Vohra, 2016), is a columnar storage format that is widely used in the big data ecosystem. It is designed to store structured tabular data, and therefore it is not ideal for storing images or nested structures. On top of that, shuffling is not part of the format and must be implemented by loading data into memory first. When it comes to compression, Parquet compresses data at the column chunk level, which means that when shuffling or sorting rows, you generally need to decompress and load the entire dataset or large chunks of it, making it rather slow and memory intensive

(Ivanov and Pergolesi, 2019).

The Hugging Face Datasets library is a powerful tool for accessing and managing machine learning datasets, particularly for natural language processing (Lhoest et al., 2021). Under the hood, the library uses Apache Arrow to enable fast access, lazy loading, and memory-efficient processing. Iteration is lazy and memory-efficient: only the accessed slices are loaded into memory. Shuffling is implemented using an index-level permutation mechanism, which avoids shuffling the actual data and instead reorders access (Hugging Face Team, 2025b). This design makes it fast and scalable, even for large datasets. Sorting is also supported through the API and involves sorting the relevant Arrow column and reindexing the dataset accordingly. However, while the library excels with text data and has growing support for general ML use cases, it is still primarily text-focused and may not offer the best performance or flexibility for large-scale multimodal data. Additionally, it lacks disk-based shuffling and sorting, meaning these operations must fit entirely in memory (Hugging Face Team, 2025a) – which becomes a bottleneck for very large datasets. Although using its streaming mode avoids loading the full dataset into memory, it does not support global shuffling or sorting, limiting flexibility for large-scale pre-processing and training workflows.

3 Innovations underlying MLDataForge

We now present the key innovations that enable the results of MLDataForge. We can divide these between new readers and writers for MDS files, a new file format, Jsonl Indexed 'N' eXtended (JINX), and its corresponding readers and writers. In addition to these innovations, MLDataForge features conversion, transformation, joining, and splitting operations, as well as support for other dataformats such as Parquet, JSONL, and MSGPACK.

3.1 MDS Compression and Efficient Reading

MDS sample compression. MDS supports both random access via its header index and transparent file-level compression. Unfortunately, combining these requires decompressing the file, increasing peak storage requirements and time consumption considerably. We implemented sample-level compression for MDS files. Unlike existing approaches that compress entire shards, sample compression operates at the individual sample level. This means only the actual data is compressed, while the index remains uncompressed. As a result, random access and operations such as iteration, global shuffling, or sorting can be performed without decompressing the shard files. Indices remain directly accessible, enabling faster and more efficient data handling and voiding the need to decompress the files before access. MLDataForge provides both a writer for producing sample compressed MDS and a drop-in replacmenet reader that can be used with MosaicML Streaming[1].

MDS bulk reader. The bulk reader is a new implementation optimized for fast, sequential reading of MDS files. Unlike traditional readers that may repeatedly open files and seek to specific positions for each sample and its index, the bulk reader loads the entire index into memory once and keeps the file open throughout the reading process. This means that when a sample is requested, the reader can quickly locate its length using the in-memory index and read it directly, without the need to reconstruct file paths or perform additional seek operations. By maintaining an open file handle and tracking the current position, the bulk reader enables faster access when reading samples in order. Its limitation is that it does not support random access, and thus neither global shuffling nor sorting.

MDS RAM reader. The RAM reader enables efficient random access across large MDS datasets, even when spread over thousands of shards. To locate a specific sample, it first uses bi-section of the cumulative sample counts to determine which shard the target index belongs to. It then consults per-shard indexes to find the exact byte offset of the sample within the file. Since the access pattern is random and we do not know in advance which shard will be needed, all shard files are memory-mapped. This allows the system to open and access many files simultaneously through virtual memory, without loading them into RAM unless required. Thus, datasets of virtually unlimited size can be accessed quickly and flexibly. Although the bulk reader offers faster performance for purely sequential reading, the RAM reader is optimized for high-speed random access at scale and still systematically outperforms the original MDS shard reader.

[1]Just add the following three lines:
```
from streaming.base.format import _readers
from mldataforge.mds import MDSSampleReader
_readers["mds"] = MDSSampleReader
```

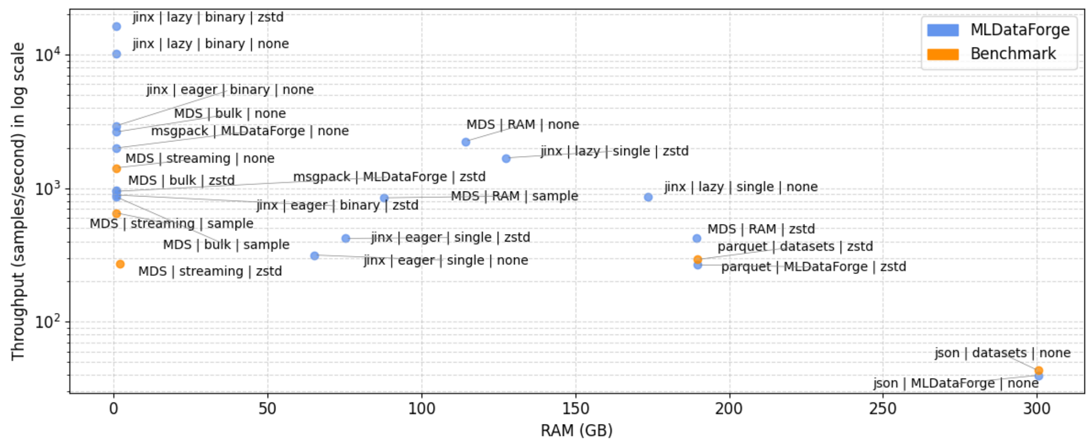

Figure 2: Throughput (samples per second) for the **iteration** operation per file type, data reader, and compression method plotted against RAM. Orange dots represent benchmark performance (our baseline), while blue dots show the results achieved with MLDataForge. Throughput is displayed on a logarithmic scale to enhance visualization clarity. The best combination (top-left) is our proposed JINX format with lazy loading, binary sidecar, and zstd compression.

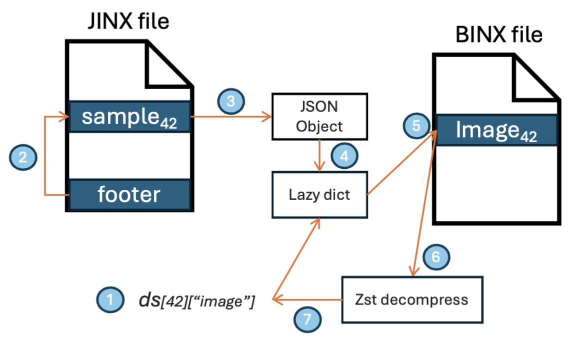

Figure 3: Workflow illustrating the steps involved when accessing a sample of the dataset, in this case sample 42. This example is for a JINX file with binary sidecar and zst compression, where *ds* is the dataset. **1-** Access the image in sample 42 of the dataset, **2-** In the JINX footer, the index specifies where to locate the offset for sample 42, **3-** JSON decode sample 42, **4-** Wrap JSON in lazy-loading dictionary, **5-** Access the image stored in the BINX file, **6-** Decompress the image data, **7-** Return the image

3.2 JSONL Indexed 'N' eXtended (JINX)

JINX is a new file format similar to JSONL, but with two additional JSON lines to support faster access. At the end of the file, it includes first a JSON object as a footer, which contains meta data and an index pointing to the start of each sample. The final line of the file is a JSON integer indicating the byte offset of the footer. As all lines are valid JSON, JINX files are technically special cases of JSONL and share human readability properties.

JINX has built-in support for handling binary content by encoding it using base64 (Josefsson, 2006). To indicate additional information about a value—such as whether it has been compressed—JINX uses special suffixes (referred to as "extensions") in the key names. For example, if a key originally named `text` holds a string that has been compressed using gzip and then encoded in base64, it would be renamed to `text.gz`.

In this way, the key name itself communicates the transformation applied to the data. Tools like MLDataForge can take advantage of this system by automatically compressing large values and renaming the keys accordingly, based on user-defined size thresholds.

Sidecar offload. Sidecar offloading is a mechanism designed for JINX files that contain large-scale data such as images or long texts. Given a value that exceeds the binary threshold, if set, it is stored in binary format in a separate binary "BINX" file. Instead of embedding the data directly through base64 encoding, the JINX file still includes critical metadata such as offset and size of the value, allowing efficient access without bloating the main file and without decreasing its human readability. Therefore, when using only the JINX file, we refer to it as a JINX "single" file. In contrast, when binary data is offloaded to a sidecar file, we refer to it as JINX "binary". In figure 3, we can see the workflow between JINX and BINX files, when accessing a sample of the dataset.

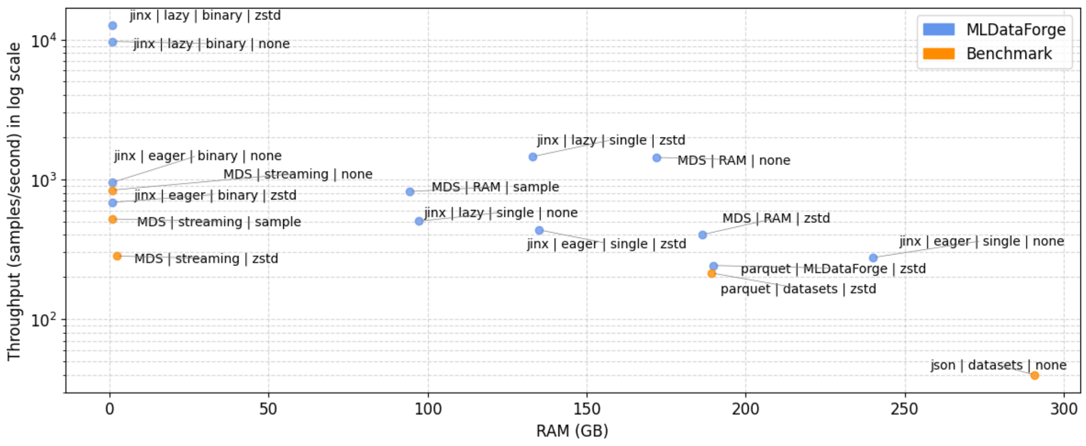

Figure 4: Throughput (samples per second) for the **shuffling** operation per file type, data reader, and compression method plotted against RAM usage. Orange dots represent benchmark performance (our baseline), while blue dots show the results achieved with MLDataForge. Throughput is displayed on a logarithmic scale to enhance visualization clarity. The best combination (top-left) is our proposed JINX format with lazy loading, binary sidecar, and zstd compression.

Lazy loading. With lazy loading for JINX and BINX files, samples are returned as lazy dictionaries whose values are only decompressed and decoded, when the corresponding key is accessed. If a binary sidecar is used, the entire load is deferred until the value is to be accessed. For instance, if an image is compressed, it will not be decompressed unless explicitly required. This approach enables operations like sorting by metadata fields without the overhead of loading large and complex data, for more efficient, on-demand access.

4 Experimental Setup

4.1 Multimodal Dataset

We conducted our experiments using the dataset `llava-instruct-mix-vsft`, a multimodal dataset available on the HuggingFace hub[2] covering text and image data. We chose this version because it adopts a modern messages structure. The dataset consists of individual entries, each containing a set of messages (a chat) and a corresponding image in JPEG format. It is organized into two subsets: a larger one with 259,155 entries called train, and a smaller one with 13,640 entries called test. When decoded and uncompressed, the train and test subsets occupy 198 GB and 10.2 GB disk space, respectively.

[2]https://huggingface.co/datasets/HuggingFaceH4/llava-instruct-mix-vsft

4.2 File Formats

The data formats used for our experimental evaluation are: MDS, JSONL, Parquet, MSGPACK, and JINX. We evaluate JINX both as single files and with binary sidecar files. The single JINX files encode binary data in base64 format rather than base85 for optimal throughput performance at a slight storage cost. These formats were obtained by converting the original files from the original multimodal dataset using MLDataForge's conversion and transformation operations. The same transformation function was applied for all formats except for JSONL, which required an encoding of the binary image data.

The MDS files where processed directly with our drop-in streaming reader that supports sample compression but otherwise is a straight copy of the original implementation, with our bulk reader, and our RAM reader. The JINX files were loaded using both the lazy and eager methods described in Section 3.2. Parquet and JSONL files were also loaded using MLDataForge, which internally relies on the HuggingFace datasets library for these two formats. Finally, we applied three compression types, when supported: no compression, Zstandard (zstd) file-level compression, and zstd sample compression (also detailed in Section 3).

4.3 Experimental Procedure

We ran two sets of experiment: one using the smaller test subset with repeated measurements,

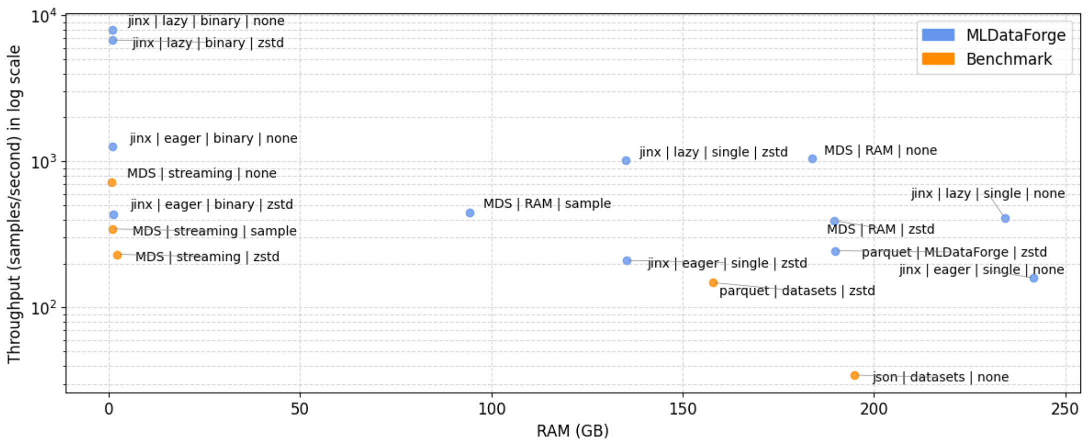

Figure 5: Throughput (samples/second) for the **sorting** operation per file type, data reader, and compression method plotted against RAM usage. Yellow bars indicate the benchmark performance, while blue bars represent the performance achieved using MLDataForge. The best combination (top-left) is our proposed JINX format with lazy loading, binary sidecar, and zstd compression.

and one set of experiments using the larger train subset of the data to demonstrate the scalability of our implementations in MLDataForge. We compared the performance of our algorithm against MosaicML Streaming and Huggingface Datasets, evaluating the time and storage requirements for iteration, global shuffling, and sorting operations. These three operations are common in data preprocessing, with the first being straightforward enough to work on all formats and the two others more challenging as global shuffling affects the order of the data access and sort likewise requires a global perspective. For each operation, we measure CPU time, system time, peak RAM usage, and peak storage usage. For the experiments on the smaller test subset, each experiment was repeated five times per file type, reader variant, and operation. For each configuration, we report the median value of these runs, as well as the interquantile range [IQR] (Frery, 2023) , as a measure of deviation. The IQR is calculated as the the difference between the 75th percentile and the 25th percentile.

We used a Linux machine with 64 CPU Cores and 384 GB of memory. These are part of a shared high-performance computing facility, and it is for this reason that we decided to repeat the experiments 5 times during the initial test and report the median to account for potential variability in performance and eliminate outliers.

4.4 Operations

Iteration. For the iteration operation, a simple in-order iteration over all samples of the dataset was carried out.

Global shuffling. For the global shuffle operation, we shuffle the indices of a dataset in a reproducible way (using a fixed seed). We then iterate over the dataset in that shuffled order.

Sorting. For the sorting operation, we sort the dataset based on a custom sorting key function applied to each sample in the dataset. We then iterate over the dataset in the resulting sorted order.

5 Results

We have collected results for all three operations (iteration, shuffling, and sorting) on both the train and the test split. From these raw results, we made two types of visualizations: a bar plot illustrating the throughput of each method (samples processed per second), and a scatter plot showing the relationship between throughput and memory usage.

In the bar plot, longer bars indicate better performance. In the scatter plot, points closer to the upper-left corner are better at balancing high throughput with low memory consumption. In both cases, the throughput has been calculated using the wall time, which corresponds to the addition of CPU time and System time. Note that the bar plot (1) displays all operations in a single figure, while the scatter plots (Figures 2, 4, and 5) are separated by operation.

We provide the full results for the iteration operation in Table 1, reporting all considered performance metrics (CPU time, system time, memory usage, and storage usage) for both the training and test splits across all evaluated file formats, readers, and compression types (see Section 4.2). For brevity, the results for sorting and shuffling are only presented in the bar and scatter plots.

5.1 Iteration

Table 1 and Figures 1 and 2 present the results of the iteration operation. While Figure 1 offers an intuitive visualization of the superiority of our approach in terms of speed, showing that the JINX binary file with lazy loading outperforms all other formats, Figure 2 presents the advantage also in terms of memory usage, both for compressed and uncompressed files.

When comparing MDS files from the benchmark (streaming reader) to our readers, our approach demonstrates a better performance. In terms of balance between throughput and memory, MDS sample compressed with Bulk reader outperforms MDS Streaming with any type of compression. In fact, our sample compression method achieves better results than zstd compression in all cases, using less system time and storage. It is also evident that HuggingFace Datasets exhibits considerably lower performance compared to the other methods.

The JSONL format was the slowest overall, with processing times exceeding 24 hours for zstd-compressed test files. The reason for this lies in the far-from-optimal encoding of the binary image data and the resulting storage pressure, which is aggravated by the HuggingFace Datasets libary's creation of multipel cached versions.

5.2 Shuffling

Figures 1 and 4 show results consistent with those observed for the iteration task. Again, the JINX binary format with lazy loading leads by a substantial margin, followed by the MDS format. While the MDS RAM variant outperforms MosaicML Streaming in terms of speed, it comes at the cost of higher RAM memory usage. Notably, both Parquet and JSON files use HuggingFace Datasets and perform rather poorly.

5.3 Sorting

Figures 1 and 5 show results consistent with the previous two operations. While sorting is the slowest operation overall, the relative performance across formats remains similar. The JINX binary format with lazy loading once again leads, with the eager loading variant of JINX binary coming in second.

6 Discussion and Conclusion

Our results across three common data access operations – iteration, shuffling, and sorting – demonstrate a clear and consistent improvement of the JINX binary format, particularly with lazy loading. This format delivers the highest throughput across all tasks but also shows substantial gains in resource efficiency, with compressed files requiring up to three times less RAM and storage compared to the MosaicML Streaming format. This is insofar expected, as the MDS format requires us to decompress full files to access the compressed index. Likewise, binary lazy loading with compression has better results than its non-compression mode, including a higher throughput. This is because lazy loading, as opposed to eager loading, does not require decompressing the binary sidecar, thus the amount of data it needs to read is determined solely by the size of the main JINX file.

Notably, the memory (RAM) usage is lower when eager loading is used compared to lazy loading. While eager loading loads all data once in a single structure, lazy loading creates new wrapper objects at all layers of the structure and each time data is accessed. Without memory pressure on the test machine forcing garbage collection, the reported RAM usage is higher for lazy loading.

For random access, our MDS RAM variant offers faster speeds than MosaicML Streaming but does so at the cost of a higher reported memory consumption. This is an artifact of the memory mapping used by the RAM reader and the memory availability on the experiment machines.

Our MDS bulk reader being faster and using less memory than the existing MosaicML Streaming reader when reading sequentially. In contrast, formats relying on HuggingFace Datasets such as Parquet and JSONL consistently underperform across all metrics, indicating that those are not ideal for high-throughput or resource-constrained environments. This is due to the fact that despite HuggingFace Datasets being able to do partial sorting quickly, they need to load full columns into memory. Hence, the library is neither designed nor particularly suited to sorting and globally shuffling full datasets.

When it comes to compression, our sample-

File	Reader	Compression	test measures				train measures			
			CPU time (seconds)	System time (seconds)	RAM (MB)	Storage (GB)	CPU time (seconds)	System time (seconds)	RAM (MB)	Storage (GB)
MDS	Streaming	none	2.17 [0.06]	5.99 [0.11]	756 [1.68]	9.74 [0.00]	37.73	119.24	898	188.77
		zstd	15.91 [0.22]	23.93 [0.81]	2,278 [1.56]	14.67 [0.00]	337.54	521.11	2,263	282.19
		sample	15.05 [0.36]	3.36 [0.16]	756 [11.70]	4.94 [0.00]	282.34	65.09	**807**	93.67
	Bulk	none	1.36 [0.08]	5.49 [0.48]	754 [5.74]	9.74 [0.00]	23.70	74.46	871	188.77
		zstd	11.83 [0.07]	3.03 [0.12]	751 [7.54]	**4.92 [0.00]**	224.78	42.45	877	188.77
		sample	13.42 [0.4]	2.85 [0.27]	760 [7.54]	4.94 [0.00]	255.27	45.34	889	**93.67**
	RAM	none	3,02 [0.36]	3,07 [0.02]	10,459 [10.13]	9.74 [0.00]	51.29	63.91	114,310.50	188.77
		zstd	14.31 [0.52]	9.15 [0.2]	10,454 [7.03]	14.67 [0.00]	285.20	309.99	189,453	282.19
		sample	13.76 [0.3]	1.94 [0.13]	5,645 [6.00]	4.94 [0.00]	267.52	38.23	87,863	93.67
JINX single	Eager	none	30.57 [0.24]	4.21 [0.03]	13,760 [0.52]	12.99 [0.00]	608.32	211.45	65,270	251.58
		zstd	28.12 [0.06]	2.30 [0.02]	7,818 [2.96]	7.08 [0.00]	553.10	62.82	75,453	134.41
	Lazy	none	10.01 [0.04]	4.67 [0.24]	13,739 [1.45]	12.99 [0.00]	201.49	96.80	173,633	251.58
		zstd	5.16 [0.17]	2.55 [0.21]	7,834 [6.93]	7.08 [0.00]	99.67	54.01	127,579	134.41
JINX binary	Eager	none	**0.40 [0.01]**	4.11 [0.22]	729 [0.29]	9.74 [0.00]	10.38	77.87	920	188.77
		zstd	12.27 [0.63]	2.36 [0.14]	741 [6.21]	5.31 [0.00]	245.15	46.17	1,029	100.88
	Lazy	none	0.42 [0.07]	1.16 [0.12]	**695 [8.55]**	9.74 [0.00]	12.24	13.07	935	188.77
		zstd	0.39 [0.00]	**0.64 [0.04]**	713 [4.89]	5.31 [0.00]	**7.60**	**8.26**	979	100.88
Parquet	MLDF	zstd	19.57 [0.15]	28.98 [0.65]	17,468 [4.94]	15.07 [0.00]	381.97	667.15	189,748	289.81
	Datasets	zstd	20.50 [0.34]	31.45 [0.03]	17,474 [8.52]	15.07 [0.00]	375.92	618.53	189,807	289.81
JSON	MLDF	none	267.60 [1.2]	75.64 [0.27]	16,899 [37.19]	58.40 [0.00]	5,097.13	3,536.10	300,357	1,140.12
		zstd	5938.92 [85.30]	46.24 [7.21]	17,102 [307.08]	22.27 [0.00]	∞	∞	-	-
	Datasets	none	262.42 [0.50]	71.64 [3.49]	16,937 [160.36]	58.40 [0.00]	4,940.72	1,667.29	300,413	1,140.12
		zstd	5162.43 [70.98]	54.30 [3.44]	16,913 [85.98]	22.27 [0.00]	∞	∞	-	-
MSGPACK	MLDF	none	2.68 [0.12]	4.46 [0.05]	749 [6.45]	9.74 [0.00]	54.48	75.35	894	188.71
		zstd	13.37 [0.19]	2.35 [2.35]	742 [26.70]	4.98 [0.00]	234.08	39.65	872	94.41

Table 1: Iteration performance under various compression settings using test and train data. The best value in each column is in bold and red colour. ∞ was assigned to time values that exceeded 24 hours. MLDF stands for MLDataForge. For the test subset, we report the median and IQR (in square brackets) of the 5 repetitions.

level compression approach consistently outperforms file-level compression, further emphasizing the advantage of format-aware optimization strategies. Notably, JSONL in particular exhibited prohibitively long processing times when zstd-compressed, exceeding 24 hours even for our size-limited test files.

Overall, the JINX format with binary sidecar file and lazy loading stands out as the most robust and scalable solution for dataset storage and access, balancing memory and storage usage while providing an order-of-magnitude improvement in sample throughput. These results strongly support its use in pre-processing and training scenarios where performance and resource optimization are critical.

Acknowledgments

This research was supported in parts by the Danish Foundation Models (DFM) project.

References

Alejandro C. Frery. 2023. *Interquartile Range*, pages 664–666. Springer International Publishing, Cham.

Bhanu Phanindra Babu Gogula. 2025. Enhancing content indexing and customer support with jsonl and ai integration. *International Journal on Science and Technology (IJSAT)*, 16:1–10.

Hugging Face Team. 2025a. Datasetinfo. Accessed: 2025-05-19.

Hugging Face Team. 2025b. What is arrow? Accessed: 2025-05-24.

T. Ivanov and M. Pergolesi. 2019. The impact of columnar file formats on sql-on-hadoop engine performance: a study on orc and parquet. *Concurrency and Computation: Practice and Experience*, 32.

Lixia Ji, Shijie Xiao, Jingmei Feng, Wenzhao Gao, and Han Zhang. 2025. Multimodal large model pretraining, adaptation and efficiency optimization. *Neurocomputing*, 619:129138.

Simon Josefsson. 2006. The base16, base32, and base64 data encodings. Technical report.

Donald E. Knuth. 1998. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., USA.

Quentin Lhoest, Albert Villanova del Moral, Yacine Jernite, Abhishek Thakur, Patrick von Platen, Suraj Patil, Julien Chaumond, Mariama Drame, Julien Plu, Lewis Tunstall, Joe Davison, Mario Šaško, Gunjan Chhablani, Bhavitvya Malik, Simon Brandeis, Teven Le Scao, Victor Sanh, Canwen Xu, Nicolas Patry, Angelina McMillan-Major, Philipp Schmid, Sylvain Gugger, Clément Delangue, Théo Matussière, Lysandre Debut, Stas Bekman, Pierric Cistac, Thibault Goehringer, Victor Mustar, François Lagunas, Alexander M. Rush, and Thomas Wolf. 2021. Datasets: A community library for natural language processing.

MosaicML Team. 2022. Streaming datasets for efficient machine learning. Accessed: 2025-05-23.

Deepak Vohra. 2016. Apache Parquet. In *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools*, pages 325–335. Apress, Berkeley, CA.