Hiding Latencies in Network-Based Image Loading for Deep Learning

Francesco Versaci[1], Giovanni Busonera[1]

[1]*CRS4 – Center for Advanced Studies, Research and Development, Cagliari, Italy*

Abstract

In the last decades, the computational power of GPUs has grown exponentially, allowing current deep learning (DL) applications to handle increasingly large amounts of data at a progressively higher throughput. However, network and storage latencies cannot decrease at a similar pace due to physical constraints, leading to data stalls, and creating a bottleneck for DL tasks. Additionally, managing vast quantities of data and their associated metadata has proven challenging, hampering and slowing the productivity of data scientists. Moreover, existing data loaders have limited network support, necessitating, for maximum performance, that data be stored on local filesystems close to the GPUs, overloading the storage of computing nodes.

In this paper we propose a strategy, aimed at DL image applications, to address these challenges by: storing data and metadata in fast, scalable NoSQL databases; connecting the databases to state-of-the-art loaders for DL frameworks; enabling high-throughput data loading over high-latency networks through our out-of-order, incremental prefetching techniques. To evaluate our approach, we showcase our implementation and assess its data loading capabilities through local, medium and high-latency (intercontinental) experiments.

Keywords

Data loading, Deep learning, High-Throughput, Latency optimizations, Scalable storage, Image classification

1. Introduction

Over the last two decades, the rapid increase in GPU computational power has transformed the field of machine learning. This surge in processing capabilities has allowed deep learning (DL) models to manage vast datasets and conduct intricate computations with unmatched efficiency. Consequently, DL techniques have gained widespread traction across various sectors, leading to advancements in areas such as cybersecurity, natural language processing, bioinformatics, and healthcare, among others [1].

However, despite these advancements in computational power, the performance of DL systems is increasingly constrained by bottlenecks related to data movement and access. While GPUs continue to achieve remarkable gains in processing throughput, which is matched by an increase in bandwidth for networking and storage, the latencies in these areas cannot decrease at the same pace due to inherent physical limitations. These discrepancies result in significant data stalls, as the transfer of data between storage systems, memory, and processing units becomes a critical bottleneck, leading to inefficient resource utilization in DL workflows [2, 3].

Furthermore, managing the vast amounts of data and associated metadata required for DL has become an increasingly complex challenge, significantly impacting the productivity of data scientists. As datasets scale from sources such as ImageNet [4], which comprises millions of images and spans hundreds of gigabytes, to more extensive datasets like LAION-5B [5], containing billions of images and reaching hundreds of terabytes, the exponential growth in data volume places substantial strain on traditional filesystem-based approaches. These systems are often inflexible and ill-suited for dynamic data management requirements. The limitations of these approaches are particularly evident when tasks such as balancing class distributions [6], or adjusting the proportions of training, validation, and test sets, must be performed, especially when considering metadata to avoid introducing unwanted biases. The static nature of traditional filesystems hampers the ability to efficiently update or modify datasets, restricting the flexibility needed for iterative development and fine-tuning of DL models.

BigHPC2025: Special Track on Big Data and High-Performance Computing, co-located with the 4th Italian Conference on Big Data and Data Science, ITADATA2025, September 9 – 11, 2025, Torino, Italy.

francesco.versaci@crs4.it (F. Versaci); giovanni.busonera@crs4.it (G. Busonera)

Moreover, DL applications often involve datasets comprising numerous small inputs that are fully scanned and randomly permuted at each training epoch. This access pattern diminishes the benefits of caching since the data is continuously reloaded and reshuffled [7]. Additionally, high-performance computing (HPC) storage systems are typically optimized for sequential access to large files, which contrasts sharply with the small, random access patterns typical of DL workloads. As a result, parallel file systems such as GPFS or Lustre in HPC environments struggle to handle these workloads efficiently [8]. A common mitigation strategy is to maintain local copies of datasets on all compute nodes. However, this approach introduces substantial storage overhead and capacity constraints. An alternative is to partition the dataset into shards distributed across nodes, which becomes essential when the dataset exceeds the storage capacity of individual nodes. Yet, ensuring unbiased data sharding poses significant challenges [8]. This strategy also compromises the ability to perform uniform random shuffling, which can negatively impact training performance and model accuracy [9].

To address these limitations, specialized data loading systems and strategies have been developed, as elaborated in the next section. However, these methods have critical shortcomings, leaving unresolved issues such as:

- Decoupling of data and metadata, leading to potential inconsistencies;
- Inflexibility of record file formats, which impose constraints on shuffling;
- Limited support for network-based data loading, resulting in local storage overload on computing nodes.

In response to these issues, in [10] we proposed leveraging scalable NoSQL databases to store both data and metadata, with preliminary performance evaluations conducted within the DeepHealth Toolkit [11], a DL framework tailored for biomedical applications. In this work, we build upon this concept by introducing and evaluating a novel data loader. Specifically, our key contributions are as follows:

- We develop an efficient data loader implemented in C++ with a Python API, designed to integrate seamlessly with Cassandra-compatible NoSQL databases and NVIDIA DALI [12, 13]. This loader supports data loading across the network and is compatible with popular DL frameworks such as PyTorch and TensorFlow.
- We introduce out-of-order, incremental prefetching techniques that enable high-throughput data loading, even in high-latency network environments;
- We conduct a comprehensive evaluation of our approach, demonstrating its implementation and benchmarking its performance through extensive experiments in local, medium and high-latency settings, comparing it against the state-of-the-art tools.

Note that, as DALI is primarily optimized for image processing, our examples will focus on DL applications involving images; however, the techniques presented are of general applicability.

2. Background and Related Work

In this section we briefly review record file formats used in DL, followed by a summary of modern data loading tools and their tradeoffs. Finally, we introduce the NoSQL Cassandra-compatible databases leveraged by our data loader.

2.1. Record File Formats in DL

Many DL applications, such as image classification, exhibit limited temporal and spatial locality due to their scan-and-reshuffle data access patterns [7], impeding I/O optimization strategies such as block reading, which involves retrieving multiple images in a single request. To address this, various record file formats group data to artificially enforce locality, enabling more efficient prefetching and reducing file system stress. Examples include TFRecord [14], RecordIO, Beton [15], and MDS (used by MosaicML [16]).

However, optimization algorithms like Stochastic Gradient Descent and Adam require uniform shuffling of data for optimal convergence [9]. Block reading conflicts with this requirement, leading to the implementation of workarounds in data loaders. Furthermore, the primary drawback of the file-batching approach is that it further rigidifies the dataset. Writing record files is time-consuming, consumes additional storage space, and makes it even more challenging to modify datasets – an already cumbersome task when dealing with numerous files in a filesystem.

2.2. Data Loading Frameworks

Efficient data loading is critical for DL training performance. Conventional Python-based pipelines suffer from GIL-induced bottlenecks and underutilized GPUs due to copy overhead betweeen processes [17, 18] and CPU-bound preprocessing. **NVIDIA DALI** [12] addresses these issues with GPU-accelerated loading, decoding, and preprocessing. It integrates with PyTorch and TensorFlow, using an asynchronous pipeline to reduce idle GPU time. However, networked data access support is still limited and experimental[1]. **TensorFlow `tf.data`** [14] provides an efficient, highly-parallel data pipeline API within TensorFlow, supporting local and Google Cloud-based sources. The experimental `tf.data service` [19] further extends this functionality for distributed pipelines by decoupling data processing and training. It can also be utilized to enable network-based data loading, with the dispatcher and workers supplying data to the nodes responsible for model training. **MosaicML Streaming Dataset** [16] is designed for training on datasets larger than memory by streaming and buffering data dynamically. It uses a proprietary format (MDS) and metadata to support sharding and determinism. **Deep Lake** [20] offers a hybrid between data lakes and vector databases, storing various data types via a proprietary format with remote access support. Its high-performance data loader is closed-source and cloud-dependent, which may limit adoption. **MADlib** [21] takes a database-centric approach, embedding ML operations within relational database management systems (RDBMS) to reduce data movement. While recent extensions support DL (e.g., with Keras), its capabilities are nascent and limited to basic image classification with uncompressed tensors and rigid minibatch layouts.

2.3. Cassandra-Compatible Databases

Our data loader, outlined in the following section, interfaces with Apache Cassandra or ScyllaDB, two distributed NoSQL databases designed for scalability and low-latency access. **Apache Cassandra** is an open-source, Java-based system known for high availability and geographic distribution. It supports sub-10ms response times and is used at scale by companies like Netflix and Spotify. **ScyllaDB** offers a C++-based, high-performance alternative compatible with the Cassandra API. It avoids Java GC overhead and employs a shard-per-core design to achieve lower latency and more predictable performance, with adopters including Discord, Ticketmaster, and Rakuten.

3. High-performance network data loading: design and implementation

Our data loader is designed to address key challenges in deep learning workflows through the following principles:

Unified data and metadata storage: Storing data alongside its metadata improves consistency and reliability, preventing errors during updates.

Fast, scalable network access: The architecture supports high-throughput, low-latency access across varied network setups, from local clusters to cloud environments.

[1]https://github.com/NVIDIA/DALI/issues/5551

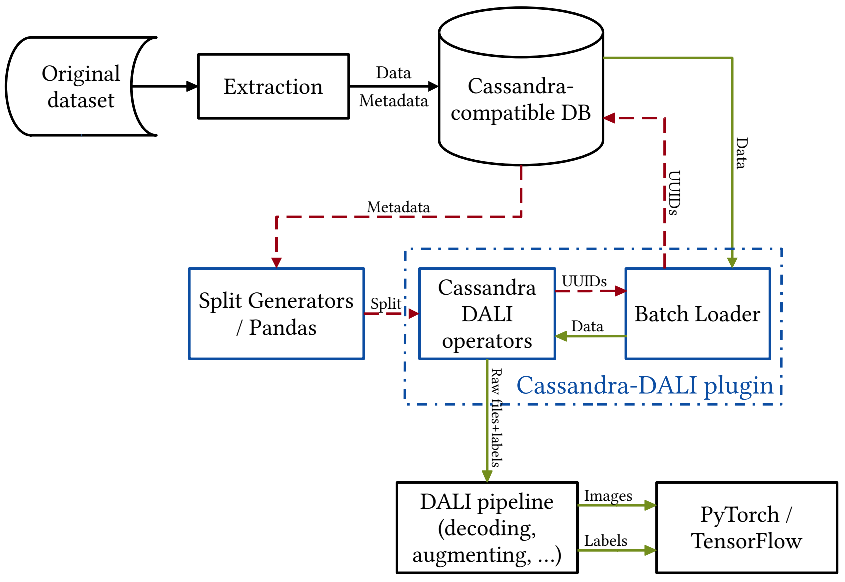


Figure 1: High-level architecture of our data loader.

Full random access: Enables efficient, unbiased shuffling by allowing retrieval of any sample, even in distributed, large-scale datasets.

Simplified data partitioning: Decouples storage from partitioning, allowing dynamic splits (e.g., cross-validation) and efficient distribution across compute nodes.

These principles underpin a robust, scalable loader tailored for large-scale DL tasks, especially image classification. To implement this, we built a plugin integrating NVIDIA DALI with a Cassandra-compatible NoSQL backend. Written in C++, it avoids Python interprocess communication overhead, significantly improving throughput and latency.

3.1. Data flow and model

Figure 1 outlines the data pipeline:

- Data and metadata are ingested into a Cassandra-compatible database.
- Each image is identified via a Universally Unique Identifier (UUID).
- Splits are defined using metadata-driven UUID lists.
- Samples are retrieved by UUID, preprocessed via DALI, and passed to the DL engine (PyTorch/TensorFlow).

This architecture enables data loading over the network using TCP, allowing full random access to datasets. Additionally, by leveraging Cassandra or ScyllaDB for storage, it offers significant advantages such as easy scalability and secure data access through SSL. It also allows for the definition of roles with detailed permissions and supports straightforward geographic replication.

As an example of image classification (which initially motivated our work) that involves a complex set of metadata, in the appendix we present SQL tables in Listing 1 that are applicable to medical imaging, specifically for tumor detection in digital pathology [22]. In this context, gigapixel images are divided into small patches, with each patch being labeled to indicate the degree or severity of cancerous tissue (Gleason score).

Our data loader supports tasks beyond standard image classification. It accepts features as generic BLOBs in any DALI-decodable format (e.g., JPEG, TIFF, PNG) and allows custom decoders. Annotations are optional and can be integers or BLOBs, supporting use cases like multilabel classification (e.g., NumPy tensors) and semantic segmentation (e.g., PNG masks).

3.2. Automatic Split Creation

Creating training, validation, and test splits requires metadata-aware strategies to ensure data independence and class balance. Entity separation – e.g., assigning patients to distinct splits – is essential to avoid data leakage and ensure generalizability. Adjusting class distributions further complicates the process.

Standard approaches, like reorganizing directories or regenerating TFRecord files, are error-prone and labor-intensive. Our plugin automates split creation and data loading, decoupling storage from partitioning. This streamlines workflows, eliminates manual overhead, and improves reproducibility, allowing users to focus on model development.

3.3. Multi-threaded asynchronous data loading

To optimize data retrieval efficiency, we leverage extensive parallelization. Images are retrieved asynchronously across multiple threads and TCP connections, thereby minimizing overall latency. Each TCP connection can handle up to 1024 concurrent requests, with the number of connections being a tunable parameter. Once the images are retrieved, batches of data are assembled in shared memory, which eliminates the need for additional copying and accelerates the process.

In detail, our batch loading workflow starts with the batch loader receiving a list of UUIDs, which it then uses to send all requests to the Cassandra driver at once. Communications for different batches are handled concurrently via a thread pool. To manage these requests efficiently, multiple low-level I/O threads are employed, each utilizing two TCP connections. Results are processed through callbacks, which minimizes latency by eliminating busy waiting. After all results for a batch have been received, the output tensor is allocated contiguously in a single operation. Data is then copied into the output tensor concurrently, again using a thread pool. The batch becomes available for output as soon as the copying is complete.

3.4. Prefetching techniques and strategies for high-latency environments

Efficient data loading is essential for optimizing training throughput, particularly in high-latency network environments. A common approach leverages the fact that, even if the dataset is shuffled at the start of each epoch, the permutation is predetermined and all the future requests are known at the beginning of each epoch. This allows to apply prefetching techniques, enabling subsequent batches to be retrieved while the GPUs process the current one, thereby minimizing idle time and improving overall efficiency.

Our data loader features a prefetching mechanism with a configurable number of batch buffers, designed to mask latencies of varying magnitudes. Despite this, during preliminary tests over real high-latency internet connections, we observed significant underperformance compared to internal tests with artificially induced latency using tc-netem [23]. Traffic analysis indicated significant bandwidth variability due to multiple TCP connections traversing different network routes, some of which experienced congestion, resulting in a wide disparity between the best and worst-performing connections. This directly impacts batch loading times: since images are retrieved in parallel but assembled in order, the system must wait for the slowest connection before proceeding, which ultimately slows down the entire process.

To address this issue, we implemented an out-of-order prefetching strategy. Given that DL training is robust to uniformly random permutations of the dataset, we can concurrently request multiple batches (e.g., 8) and reassemble them based on the arrival time of the contained images. This approach reduces the impact of slow connections by prioritizing images that arrive first. For this strategy to function correctly, it is essential that labels are retrieved together with their corresponding features, a requirement met by our architecture, which retrieves both features and annotations with a single query.

Further testing over real high-latency internet connections revealed another, second-order, issue: an aggressive filling of the prefetch buffers (e.g., 8 buffers per GPU across 8 GPUs) can cause a burst of requests that temporaly overwhelms the network capacity, leading to unstable throughput during buffer filling. To mitigate this, we can stagger the prefetch requests over time. For example, instead of front-loading all prefetch requests, we can request an extra batch every four regular ones: for every four batches consumed, five new ones are requested until the buffer is full. This approach limits the increase in transient throughput to only 25% above the steady-state level.

These optimizations collectively enhance throughput and stabilize data loading in high-latency environments, significantly improving resource utilization, as detailed in Sec. 4.

4. Software evaluation and results

4.1. Availability and usage

Our data loader is free software released under the Apache-2.0 license and is accessible on GitHub at the following URL: https://github.com/crs4/cassandra-dali-plugin/. The repository includes a Docker container that provides a pre-configured environment with DALI, Cassandra DB, a sample dataset for experimentation, and comprehensive instructions for conducting further tests. The repository provides scripts to streamline the ingestion of datasets into Cassandra, either serially or in parallel via Apache Spark. It also includes examples of multi-GPU DL training in PyTorch, using both plain PyTorch and PyTorch Lightning. Additionally, it features tools for automatic dataset splitting and high-performance inference using NVIDIA Triton[2]. This setup enables clients to request inference of images stored on a remote Cassandra server to be processed on a different GPU-powered remote server.

Listing 2 in the appendix presents the Python code used to initialize a typical DALI pipeline, which includes data loading via the standard file reader, decoding, and standard preprocessing steps such as resizing, cropping, and normalization. To use our custom data loader, which supports data reading from a Cassandra-compatible DB, one simply needs to replace the standard file reader with our module, as demonstrated in Listing 3. The modified lines are highlighted in blue.

4.2. Comparative analysis at varying latencies

We evaluated our network data loader alongside two state-of-the-art competitors, leveraging Amazon EC2 instances located in different geographical regions to introduce varying latencies. Specifically, we tested data consumption in Oregon using an 8-GPU p4d.24xlarge node, while storing images at locations characterized by the following latencies:

- **Low:** Data stored in Oregon, round-trip time (RTT) < 1 ms.
- **Medium:** Data stored in Northern California, RTT $\simeq$ 20 ms.
- **High:** Data stored in Stockholm, Sweden, RTT $\simeq$ 150 ms.

While storing images on a different continent from the GPUs is not a common practice, the high-latency scenario was included to highlight the challenges posed by latency-induced bottlenecks. Such challenges are expected to further intensify in the future, as computational power and bandwidth improve, whereas latency remains constrained by physical limits.

For our experiments, we utilized the standard ImageNet-1k dataset (training set: 1,281,167 images, average image size: 115 kB, total size: 147 GB, batch size: 512 images.) The dataset was prepared in the formats required by each data loader and stored on a single node equipped with four NVMe SSDs, configured as a single striped logical volume for optimized data access. Specifically, we used r5dn.24xlarge instances in Oregon and Stockholm, and the similar m6in.24xlarge instance in Northern California, where the previous instance type was not available. As the RAM capacity of these machines surpasses the size of our test dataset (i.e., ImageNet-1k), we reserved memory to maintain approximately only 70 GB of free RAM (i.e., half the size of the dataset). This approach prevents dataset caching in main memory, ensuring that data is consistently read from the disks during testing. By doing so, we simulate conditions involving larger datasets that exceed the available memory capacity.

Three data loaders, summarized in Table 1, were compared in this study: our Cassandra-DALI data loader (with data stored in high-performance ScyllaDB), MosaicML SD (with data hosted on an S3

[2]https://developer.nvidia.com/triton-inference-server

Table 1
Overview of the tested data loaders

Data Loader	Version	Data Storage	Granularity
Cassandra-DALI	1.2.0	ScyllaDB (Cassandra-compatible)	Single image
MosaicML SD	0.10.0	MinIO S3-compatible server	Record file (MDS)
tf.data service	2.16.1	Filesystem on remote node	Record file (TFRecord)

MinIO server), and TensorFlow's tf.data service (with data stored as TFRecords in the filesystem). All servers were hosted on the same node within Docker containers, with all data residing on the same logical volume. The test code, including the Dockerfiles, is available in the following GitHub repository, under the *paper* branch: https://github.com/fversaci/cassandra-dali-plugin/.

We conducted two experiments to evaluate performance under varying latencies:

Tight-loop read: This benchmark assesses raw data-loading capabilities by maximizing data reads without GPU processing or image decoding.

Training: A standard PyTorch multi-GPU ResNet-50 training workload, including image decoding and preprocessing steps such as resizing, normalization, and cropping.

It is important to note that the tight-loop read test utilizes a single data loader, whereas the training process employs a separate data loader for each GPU. Consequently, the tight-loop read test establishes an upper bound on the data throughput that can be consumed by a single GPU.

To minimize AWS usage costs, each data loader was evaluated in a single test run for up to four epochs. The experimental results presented below compare the performance of the data loaders under varying latency conditions.

4.2.1. Tight-loop reading

As a baseline for comparing network data loaders, we measured the performance of NVIDIA DALI when reading images stored as TFRecords from the local filesystem, without performing image decoding. This configuration achieved a data throughput of 7.4 GB/s.

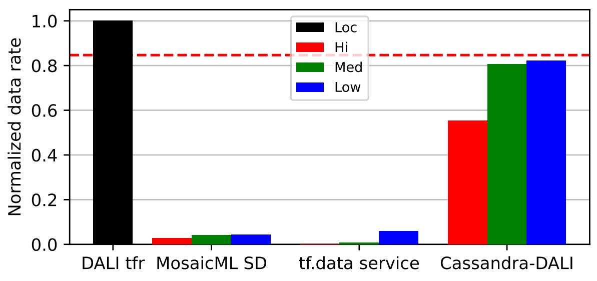
The p4d.24xlarge instance offers a total network bandwidth of 100 Gb/s; however, only half of this bandwidth is accessible through its public interface[3]. Thus, the maximum raw bandwidth available for data transfers in our tests was limited to 50 Gb/s (6.25 GB/s). As shown in Fig. 2a and Tab. 2, our data loader nearly saturated the available bandwidth when reading from ScyllaDB in both local and medium-latency settings, achieving a throughput of approximately 6 GB/s in each case. In the high-latency setting, throughput decreased to around 4 GB/s.

In contrast, MosaicML SD demonstrated significantly lower performance, with throughput measured at 326 MB/s in the low-latency setting, 308 MB/s in the medium-latency setting, and 203 MB/s in the high-latency setting. tf.data service exhibited better performance than MosaicML SD in the low-latency configuration, achieving a throughput of 437 MB/s. However, its performance degraded substantially in higher-latency environments, with throughput dropping to 57 MB/s in the medium-latency setting and just 12 MB/s in the high-latency setting.

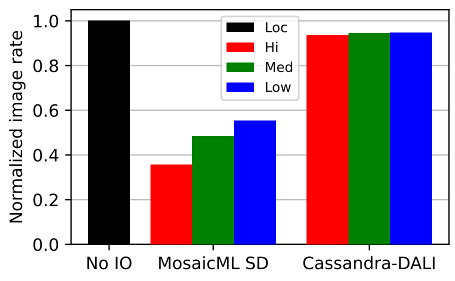
4.2.2. Training

For a data loader to be effective, its performance must integrate smoothly into the DL pipeline, ensuring that tensors are efficiently delivered to DL engines without introducing bottlenecks or delays. To evaluate this, we assessed the performance of data loaders within a standard image classification training workflow using the ResNet-50 architecture. Due to the differences in training performance between TensorFlow and PyTorch, we focused exclusively on one framework to ensure a fair comparison. Given that MosaicML SD demonstrated superior performance compared to tf.data service in medium-

[3]https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-network-bandwidth.html



(a) Normalized tight-loop reading throughput.



(b) Normalized train reading throughput.

Figure 2: Normalized reading throughput at varying latencies. The red, dashed line in (a) shows the available network bandwidth.

and high-latency settings in the previous evaluation, we chose to test it against our data loader in a training using PyTorch.

To establish a performance upper bound, we first performed training using a fixed input tensor, thereby eliminating the overhead associated with data loading and preprocessing. This setup enabled us to measure the maximum achievable data throughput during training. Specifically, we recorded the number of images processed per second on a single GPU and across all 8 GPUs during multi-GPU training. Our results indicate that a single NVIDIA A100 GPU consumes about 1450 images/s, while an 8-GPU configuration reaches 11200 images/s. Given the average ImageNet-1k training image size of 115 kB, the data loaders must sustain a steady throughput of approximately 1.3 GB/s to meet these requirements.

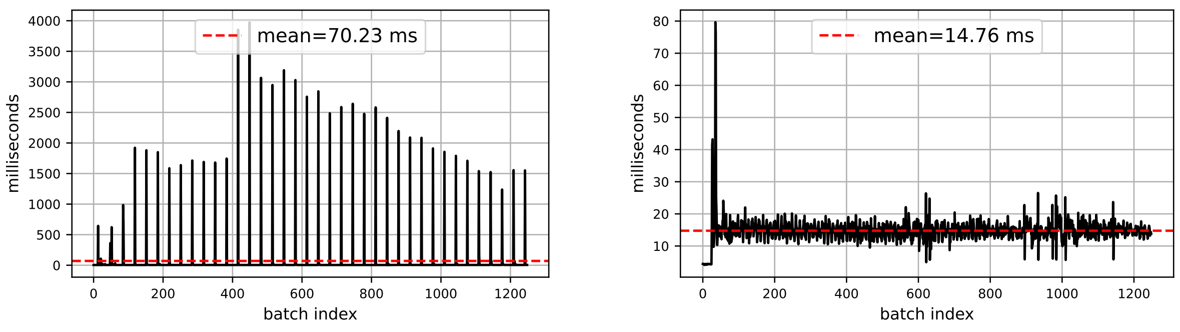
As demonstrated in Fig. 2b and Tab. 3, the MosaicML SD data loader is unable to sustain the throughput required to fully utilize all 8 GPUs, achieving 57%, 49%, and 33% of the target throughput under low, medium, and high-latency conditions, respectively. In contrast, our data loader achieves 94%, 95%, and 96% of the theoretical upper bound in these settings.

Table 2
Tight-loop reading at varying latencies. The table shows average epoch data throughput and standard deviation (omitted when data is insufficient). Epoch throughput is calculated as the dataset size divided by epoch duration.

Data loader	Throughput (MB/s)			
	Local	Low-lat	Med-lat	Hi-lat
DALI TFRecord	7381 ± 108			
MosaicML SD		326 ± 14	308 ± 15	203 ± 8
tf.data service		437 ± 5	57	12
Cassandra-DALI		6066 ± 147	5957 ± 115	4081 ± 337

Table 3
Pytorch ResNet-50 training at varying latencies. The table shows average epoch image throughput and standard deviation (omitted when data is insufficient). Epoch throughput is calculated as the dataset size divided by epoch duration.

Data loader	Throughput (img/s)			
	Local	Low-lat	Med-lat	Hi-lat
No I/O	11199 ± 312			
MosaicML SD		6209 ± 89	5424 ± 735	3992 ± 5
Cassandra-DALI		10608 ± 113	10587 ± 300	10485 ± 98



(a) In-order prefetching. (b) Out-of-order prefetching.

Figure 3: Comparison of batch loading times in high-latency networks.

4.3. Impact of prefetching and database choice on data loader performance

Finally, we conducted further tests to better investigate our data loader's performance, focusing specifically on the impact of the proposed out-of-order prefetching optimization. Additionally, we analyzed how the choice of the underlying database – either Cassandra or ScyllaDB – affects data loading performance.

4.3.1. Impact of out-of-order, incremental prefetching

We evaluated the tight-loop reading performance under high-latency conditions, comparing results with and without our out-of-order, incremental prefetching optimization.

High-latency, high-bandwidth internet communications are prone to significant variability in TCP throughput. In fact these conditions exacerbate the effects of packet loss, as TCP congestion control mechanisms respond conservatively to retransmissions and recover slowly due to extended RTTs, resulting in reduced throughput [24, 25].

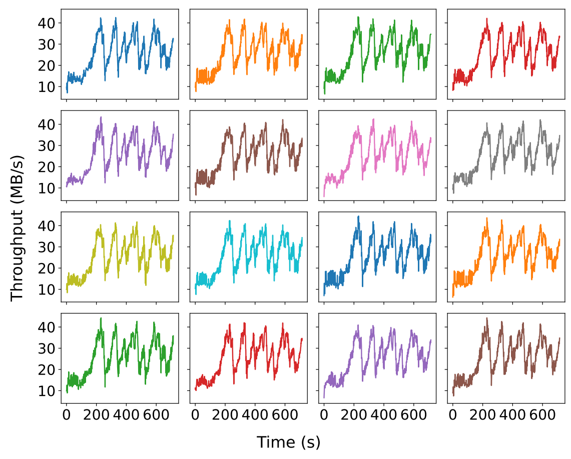
Figure 3 highlights the significant variance in batch loading times between in-order and out-of-order prefetching. In the in-order case (Figure 3a), when the prefetching queue is exhausted, the system experiences delays of up to several seconds while waiting for all transfers, including those over congested routes, to complete. This results in a cyclical pattern: once all transfers are completed, the queue is refilled, but it is quickly depleted again, triggering a new cycle. In contrast, the out-of-order approach (Figure 3b) maintains a highly consistent batch loading time, staying always below 30 ms after the initial transient period.

Figure 4a illustrates the throughput over time for 16 of the 32 connections utilized by our data loader when employing the standard in-order prefetching mechanism. The throughput curves exhibit a strong correlation, highlighting that simultaneous transfers are constrained by the in-order batch assembly process, since the system must wait for the slowest transfer to finish before dispatching a batch to the DL pipeline and requesting a new batch from the database. As a result, the throughputs tend to converge and the aggregated throughput exhibits considerable fluctuations, ranging roughly from 300 MB/s to 1300 MB/s.

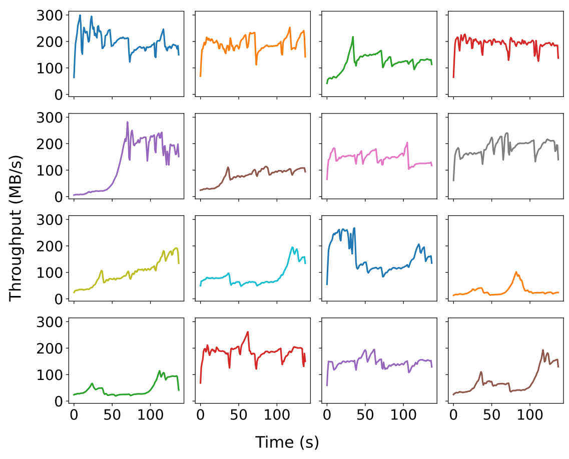
In contrast, relaxing this in-order constraint allows transfers to proceed independently, as shown in Figure 4b. In this approach, batches are formed as soon as a sufficient number of images are available, irrespective of their originating connections. This optimization significantly enhances overall throughput, resulting in higher and more consistent performance, maintaining an average throughput of approximately 4 GB/s.

4.3.2. Cassandra vs ScyllaDB

The tight-loop reading test under high-latency conditions was also performed using Cassandra as the storage backend for images, replacing ScyllaDB. Cassandra achieved a throughput of 1.6 GB/s,



(a) In-order prefetching.



(b) Out-of-order prefetching.

Figure 4: Transfer rate of 16 (out of 32) concurrent, high-latency TCP connections.

significantly lower than the 4.0 GB/s observed with ScyllaDB, highlighting the superior performance of the latter. Notably, Cassandra exhibited a disk I/O rate considerably higher than its achieved data throughput (3.6 GB/s versus 1.6 GB/s), likely attributable to differences in its block-reading strategy compared to ScyllaDB.

5. Conclusion

Advances in GPU power have propelled deep learning but also highlighted bottlenecks in data access and movement, due to the increasing discrepancy between processing throughput and data access latencies. We address these challenges by integrating scalable NoSQL databases with a high-performance, image-optimized data loader.

Our key contribution is a novel loader using advanced prefetching, including out-of-order strategies to reduce the effects of network latency. By coupling data with metadata in a database-driven design, we offer a scalable and consistent solution for DL datasets. Experiments under varying latency conditions show significant gains in throughput and stability over existing methods.

The source code for our implementation is publicly available, providing a resource for further research and practical deployment in diverse DL scenarios.

Acknowledgments

This work was supported by the Italian Ministry of Health under the program grant H2ub (*Hybrid Hub: Cellular and computation models, micro- and nano-technologies for personalized innovative therapies*, project code T4-AN-10) and by the Regione Autonoma della Sardegna, Sardegna Ricerche, under the program grant XDATA.

Declaration on Generative AI

During the preparation of this work, the authors used ChatGPT-4 for grammar and spelling check. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the publication’s content.

References

[1] L. Alzubaidi, J. Zhang, A. J. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, J. Santamaría, M. A. Fadhel, M. Al-Amidie, L. Farhan, Review of deep learning: concepts, cnn architectures, challenges, applications, future directions, Journal of big Data 8 (2021) 1–74.

[2] M. Kuchnik, A. Klimovic, J. Simsa, V. Smith, G. Amvrosiadis, Plumber: Diagnosing and removing performance bottlenecks in machine learning data pipelines, in: D. Marculescu, Y. Chi, C. Wu (Eds.), Proceedings of Machine Learning and Systems, volume 4, 2022, pp. 33–51.

[3] J. Mohan, A. Phanishayee, A. Raniwala, V. Chidambaram, Analyzing and mitigating data stalls in dnn training, Proc. VLDB Endow. 14 (2021) 771–784. URL: https://doi.org/10.14778/3446095.3446100. doi:10.14778/3446095.3446100.

[4] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, L. Fei-Fei, Imagenet: A large-scale hierarchical image database, in: 2009 IEEE Conference on Computer Vision and Pattern Recognition, 2009, pp. 248–255. doi:10.1109/CVPR.2009.5206848.

[5] C. Schuhmann, R. Beaumont, R. Vencu, C. Gordon, R. Wightman, M. Cherti, T. Coombes, A. Katta, C. Mullis, M. Wortsman, P. Schramowski, S. Kundurthy, K. Crowson, L. Schmidt, R. Kaczmarczyk, J. Jitsev, Laion-5b: An open large-scale dataset for training next generation image-text models, in: S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, A. Oh (Eds.), Advances in Neural Information Processing Systems, volume 35, 2022, pp. 25278–25294.

[6] K. Ghosh, C. Bellinger, R. Corizzo, P. Branco, B. Krawczyk, N. Japkowicz, The class imbalance problem in deep learning, Machine Learning 113 (2024) 4845–4901.

[7] J. Lee, H. Bahn, Analyzing data reference characteristics of deep learning workloads for improving buffer cache performance, Applied Sciences 13 (2023). URL: https://www.mdpi.com/2076-3417/13/22/12102. doi:10.3390/app132212102.

[8] F. Schimmelpfennig, M.-A. Vef, R. Salkhordeh, A. Miranda, R. Nou, A. Brinkmann, Streamlining distributed deep learning i/o with ad hoc file systems, in: 2021 IEEE International Conference on Cluster Computing (CLUSTER), 2021, pp. 169–180. doi:10.1109/Cluster48925.2021.00062.

[9] L. Xu, S. Qiu, B. Yuan, J. Jiang, C. Renggli, S. Gan, K. Kara, G. Li, J. Liu, W. Wu, et al., Stochastic gradient descent without full data shuffle: with applications to in-database machine learning and deep learning systems, The VLDB Journal (2024) 1–25.

[10] F. Versaci, G. Busonera, Scaling deep learning data management with cassandra db, in: 2021 IEEE International Conference on Big Data (Big Data), 2021, pp. 5301–5310. doi:10.1109/BigData52589.2021.9672005.

[11] M. Cancilla, L. Canalini, F. Bolelli, S. Allegretti, S. Carrión, R. Paredes, J. A. Gómez, S. Leo, M. E. Piras, L. Pireddu, et al., The deephealth toolkit: a unified framework to boost biomedical applic ations, in: 2020 25th International Conference on Pattern Recognition (ICPR), IEEE, 2021, pp. 9881–9888.

[12] J. A. Guirao, K. Łęcki, J. Lisiecki, S. Panev, M. Szołucha, A. Wolant, M. Zientkiewicz, Fast AI Data Preprocessing with NVIDIA DALI, https://developer.nvidia.com/blog/fast-ai-data-preprocessing-with-nvidia-dali/, 2019. [Online; accessed August-2024].

[13] E. J. Martinez-Noriega, C. Peng, R. Yokota, High-performance data loader for large-scale data processing, Electronic Imaging 36 (2024) 196–1–196–1. URL: https://library.imaging.org/ei/articles/36/12/HPCI-196. doi:10.2352/EI.2024.36.12.HPCI-196.

[14] D. G. Murray, J. Simsa, A. Klimovic, I. Indyk, tf. data: A machine learning data processing framework, arXiv preprint arXiv:2101.12127 (2021).

[15] G. Leclerc, A. Ilyas, L. Engstrom, S. M. Park, H. Salman, A. Mądry, Ffcv: Accelerating training by removing data bottlenecks, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2023, pp. 12011–12020.

[16] T. M. M. Team, streaming, <https://github.com/mosaicml/streaming/>, 2022.

[17] R. Meier, T. R. Gross, Reflections on the compatibility, performance, and scalability of parallel python, in: Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages, DLS 2019, Association for Computing Machinery, New York, NY, USA, 2019, p. 91–103.

URL: https://doi.org/10.1145/3359619.3359747. doi:10.1145/3359619.3359747.

[18] A. Ho, [rfc] polylithic: Enabling multi-threaded dataloading through non-monolithic parallelism, https://github.com/pytorch/data/issues/1318, 2024.

[19] A. Audibert, Y. Chen, D. Graur, A. Klimovic, J. Šimša, C. A. Thekkath, tf.data service: A case for disaggregating ml input data processing, in: Proceedings of the 2023 ACM Symposium on Cloud Computing, SoCC '23, Association for Computing Machinery, New York, NY, USA, 2023, p. 358–375. URL: https://doi.org/10.1145/3620678.3624666. doi:10.1145/3620678.3624666.

[20] S. Hambardzumyan, A. Tuli, L. Ghukasyan, F. Rahman, H. Topchyan, D. Isayan, M. McQuade, M. Harutyunyan, T. Hakobyan, I. Stranic, et al., Deep lake: A lakehouse for deep learning, in: Conference on Innovative Data Systems Research (CIDR 2023), 2023.

[21] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, et al., The madlib analytics library, Proceedings of the VLDB Endowment 5 (2012).

[22] S. Banerji, S. Mitra, Deep learning in histopathology: A review, WIREs Data Mining and Knowledge Discovery 12 (2022) e1439. URL: https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/widm.1439. doi:https://doi.org/10.1002/widm.1439. arXiv:https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/widm.1439.

[23] S. Hemminger, et al., Network emulation with netem, in: Linux conf au, volume 5, 2005, p. 2005.

[24] D. Katabi, M. Handley, C. Rohrs, Congestion control for high bandwidth-delay product networks, SIGCOMM Comput. Commun. Rev. 32 (2002) 89–102. URL: https://doi.org/10.1145/964725.633035. doi:10.1145/964725.633035.

[25] S. Ha, I. Rhee, L. Xu, Cubic: a new tcp-friendly high-speed tcp variant, SIGOPS Oper. Syst. Rev. 42 (2008) 64–74. URL: https://doi.org/10.1145/1400097.1400105. doi:10.1145/1400097.1400105.

A. Listings

Listing 1 Example of SQL data model for tumor detection

```
CREATE TABLE patches.metadata(
  patient_id text,
  slide_num int,  // patients can have several slides
  x int,  // coordinates
  y int,  // within the slide
  label int,  // Gleason score
  patch_id uuid,
  PRIMARY KEY ((patch_id))
);

CREATE TABLE patches.data(
  patch_id uuid,
  label int,  // Gleason score
  data blob,  // image/tensor file (JPEG, TIFF, NPY, etc.)
  PRIMARY KEY ((patch_id))
);
```

Listing 2 Initializing DALI pipeline using DALI standard file reader

```
@pipeline_def(batch_size=128, num_threads=4, device_id=device_id)
def get_dali_pipeline():
    images, labels = fn.readers.file(name="Reader",
      file_root="/data/imagenet/train")
    labels = labels.gpu()
    images = fn.decoders.image(images,  device="mixed",
      output_type=types.RGB)
    images = fn.resize(images, resize_x=256, resize_y=256)
    images = fn.crop_mirror_normalize(images, dtype=types.FLOAT,
        output_layout="CHW", crop=(224, 224),
        mean=[0.485 * 255, 0.456 * 255, 0.406 * 255],
        std=[0.229 * 255, 0.224 * 255, 0.225 * 255],
    )
    return images, labels
```

Listing 3 Initializing DALI using our Cassandra-DALI plugin

```
uuids = list_manager.get_list_of_uuids(...)

@pipeline_def(batch_size=128, num_threads=4, device_id=device_id)
def get_dali_pipeline():
    images, labels = fn.crs4.cassandra(name="Reader",
        cassandra_ips=[1.2.3.4, 5.6.7.8],
        username="guest", password="test",
        table=imagenet.data_train, uuids=uuids,
        prefetch_buffers=16, io_threads=8
    )
    labels = labels.gpu()
    # [...] same as before
    return images, labels
```