Scaling deep learning data management with Cassandra DB[*]

Francesco Versaci and Giovanni Busonera

CRS4, Cagliari, Italy

January 19, 2022

Abstract

Deep learning (DL) algorithms require, to be fully effective, harvesting an increasingly large amount of data. These data, typically organized as millions of small files, stress filesystems and are difficult to manage. In fact, despite the huge development of DL tools and specialized hardware, data loading pipeline for DL still lacks behind in ease of use, standardization and scalability.

In this work we try to rethink the data loading pipeline, by leveraging NoSQL DBs for storing both data and metadata, making them efficiently available through the network, and allowing easier data distribution for parallel DL training. We present our open-source, Apache Cassandra-based data loader and illustrate its use and performance, which enable easy and efficient data management and decentralized data distribution for parallel learning applications.

1 Introduction

Deep learning (DL) techniques are now ubiquitous and have been adopted in countless applications, and, thanks to ever more powerful GPUs and accelerators, they produce increasingly accurate predictions. In order to be fully effective, DL algorithms require processing an increasingly large amount of data that can easily comprise millions of different files and the associated metadata. However, while a lot of effort has been spent in optimizing the DL computational process, the data-loading pipeline still lacks behind in ease of use, standardization and scalability [8, 30, 33].

As an example, let us consider the key problem of image classification, and tissue classification in particular, an important problem in Digital Pathology (e.g., the automated classification of breast or prostate tissue [20, 27]). The input dataset consists of hundreds or thousands of gigapixel images (slides), from which smaller portions (patches) are extracted, together with their labels (e.g., normal/tumor), to make up the dataset for the DL training. Each patch has

[*]The final authenticated version of this article is available online at `https://doi.org/10.1109/BigData52589.2021.9672005`

complex metadata that need to be tracked (coordinates within the slides, patient id, date, etc.) and which also need to be taken into account when creating train, test and validation sets. For example, patches must be divided into splits (e.g., train, validation, etc.) according to the patient id, and some target balance between classes in the training dataset (i.e., labels) is usually desired (e.g., 1:1 normal/tumor).

The typical workflow [16] requires some custom program to build the DL dataset, which will consist of millions of small files, saved in a filesystem, with their path encoding the split and class to which they belong (e.g., `train/normal`, `validation/tumor`, etc.). This process presents many drawbacks, affecting both usability and performance:

- The dataset is static and if some changes are required (e.g, a different train/validation ratio) this could imply recreating or moving millions of files.
- The user needs a custom way to keep track of the metadata (e.g., a database or a CSV file), since he might need them at later stages of the processing.
- To allow parallel access to the dataset (e.g., for a distributed training or for different trainings working on the same dataset) one needs to either move the data to a network storage (which would decrease dramatically the data-loading performance) or set up a parallel filesystem, which is a complex task requiring dedicated hardware and careful configurations, while still often underperforming when accessing small files [5, 13, 19, 21].

In this work we show how these problems can be addressed by adopting a data management strategy based on NoSQL databases, which can be leveraged to achieve horizontal scalability and low-latency access to the training data and metadata, via a user-friendly, flexible interface.

To illustrate our strategy we have developed an Apache Cassandra-based [17] data-loading module, which is focused on image classification and is being integrated with the DeepHealth Toolkit [4]. However, its design and architecture are of general interest and applicability.

The contributions are summarized as follows:

- We present a scalable strategy, based on Cassandra DB, to easily and efficiently manage data and metadata for DL.
- We describe a newly developed data loader, which implements our proposed management strategy, and analyze its performance and use.
- We extend the EDDL library [4], using Message Passing Interface (MPI), to support synchronized data parallelism.
- We show how our data loader can be used to easily distribute data, in a decentralized way, among the workers participating in a distributed training.

The rest of this manuscript is structured as follows. Section 2 provides some technology background. In Sec. 3 we describe the high-level design of our data-loading module, while its implementation is detailed in Sec. 4. Section 5 presents and discusses the empirical performance of the data loader, and Sec. 6 focuses in particular on distributed training. Finally, Sec. 7 points the reader to the software and Sec. 8 concludes the manuscript.

2 Background

2.1 Related work

Within the TensorFlow framework, an approach to mitigate the small file problem is provided by TFRecord [28], which is a serialization format that allows for efficient loading and saving of datasets to/from disks, by grouping more records into bigger files, and also allows the saving of labels along with the features. However, it does not support random access to the saved items (and hence, e.g., global shuffling of the dataset), nor it removes the disk and network bottlenecks when accessing the data.

An interesting approach for dealing with parallel filesystem bottlenecks in a distributed context has been adopted in [14]: instead of allowing full access to the dataset, each node is allowed to view only a subset of the data and the authors have developed a custom, multi-threaded data stager which reads partitioned data from GPFS and distributes them to the computing nodes via MPI calls. Yet, this procedure amounts to a static pre-distribution of data and hence cannot handle image metadata nor it can support random access to the images.

FanStore [32] presents a more general approach to overcome filesystem bottlenecks, by developing a custom, MPI-based, parallel filesystem, which exposes a POSIX interface, and is tailored for DL application. It offers good scalability, but since it exposes a filesystem interface, is not designed to easily handle image metadata.

A further step in overcoming parallel filesystem bottlenecks by completely avoiding communications can be seen in [15], where instead of transferring the data needed for the training, they are generated in real-time in each computing node. This technique, however, can be applied only in limited contexts, in which training is run on synthetic data.

There have also been approaches connecting DL frameworks to key-value DBs (e.g., the *ml-pyxis* plugin for PyTorch, which leverages the Lightning Memory-Mapped Database [22]), but they lack the scalability and flexibility of the data loader presented in this paper.

2.2 DeepHealth Toolkit

The DeepHealth Toolkit [4] is an open-source DL toolkit, particularly focused on enabling easy DL adoption in the medical field. It is written in C++, exposes C++ and Python APIs, and it natively supports cloud computing. Our data-loading module is written to interface with the DeepHealth Toolkit, but it may be adapted, without too much effort, to work with other popular DL frameworks, such as TensorFlow or PyTorch.

2.3 Apache Cassandra

NoSQL databases are data storage systems which support high availability and horizontal scalability, at the expense of lower consistency guarantees than standard SQL databases [7]. Apache Cassandra [17] is a distributed, decentralized and highly scalable NoSQL DB, it is a free and open-source project and is widely adopted both in industry (e.g., Netflix, Uber [6]) and big data analytics contexts [25] (e.g., in the CERN ATLAS project [26]). As for the performance, Apache

Cassandra offers low-latency (typically less than a millisecond), high-bandwidth, concurrent accesses to the stored data, while supporting easy scalability, high availability and tunable data redundancy. Also, it does not require special hardware (apart from fast disks and/or large memory) and thus it could even be installed on the very same computing nodes, if needed.

3 Architecture

Our effort in rethinking the data loading pipeline is twofold, having both usability and performance objectives:

- we aim at storing and accessing features and metadata uniformly, i.e., using the same system (Cassandra DB in our case) for both;
- we want to allow scalable, flexible and fast network access to the data;
- we aim at offering dynamic, random access to the data, allowing full, unrestricted access to the dataset (since every DL training algorithm based on Stochastic Gradient Descent needs that data is sufficiently – ideally uniformly – reshuffled after every epoch [10]);
- we want to decouple storage and splits management, so that different datasets (also insisting on the same set of images) may be used, also concurrently, whenever needed (e.g., initial tests can be run on a smaller subset of data, and subsets with different characteristics can easily be obtained and explored by filtering according to metadata);
- we want to simplify the data distribution in parallel DL training.

3.1 Workflow

The workflow that we have designed to achieve the previous objectives is the following:

- All the images that might be needed for DL are saved as BLOBs in the Cassandra DB (details in 3.2), together with labels and metadata, and are identified by a UUID, thus allowing collision-free, distributed, uncoordinated data insertion [18].
- The DB is queried to get the full list of UUIDs and the metadata which are required for creating the splits.
- The splits (expressed as lists of UUIDs) are then created automatically, based on target values and constraints involving metadata (see 4).
- Finally, during the training and validation phases, when the data are needed they are efficiently pre-fetched by their UUID, and fed to the DL library.

3.2 Data model

When designing data models for NoSQL databases, particular attention must be devoted to choosing which keys to adopt and which tables to denormalize, since from these initial choices it will depend which queries will be allowed and how the system will perform when answering them. To allow both a fast retrieval of data as well as an easy access to metadata, we have chosen to organize datasets in three tables:

metadata_by_nat Each record of this table contains all the metadata, the label, and a randomly generated UUID. Its partition keys are the "natural" ones of the dataset, plus the label (see example below).

data_by_uuid Records in this tables contain only the minimum data needed by the training, i.e., the BLOB of the image files, the label, and finally the UUID as primary key.

metadata_by_uuid This (optional) denormalized table contains all the fields of metadata_by_nat, but it has the UUID as primary key.

This data organization is extremely flexible and can easily be adopted in most image classification contexts.

3.3 CQL tables

The user has to identify the required metadata for each dataset that wants to use, and create the appropriate tables to store data and metadata in the Cassandra DB. The list of columns is then passed to the data loader (see Sec. 4 and the example in Listing 1), that will accordingly use these columns when creating the splits.

As an example, here is a minimal CQL description of the tables that might be needed for the automated tissue classification.

```
CREATE TABLE tissue_patches.metadata_by_nat(
  patient_id text,  // e.g., P1234
  slide_num int,    // e.g., 5 (out of 10, for this patient)
  x int, // x coordinate within slide
  y int, // y coordinate within slide
  label int, // e.g., 0 = normal, 1 = tumoral
  patch_id uuid,    // e.g., 3d50c252-9e14-47c3...
  PRIMARY KEY  ((patient_id, slide_num, label), x, y)
);

CREATE TABLE tissue_patches.data_by_uuid(
  patch_id uuid,
  label int,
  data blob, // image file (JPEG, TIFF, etc.)
  PRIMARY KEY ((patch_id))
);

CREATE TABLE tissue_patches.metadata_by_uuid(
  patient_id text,
  slide_num int,
  x int,
  y int,
  label int,
  patch_id uuid,
  PRIMARY KEY ((patch_id))
);
```

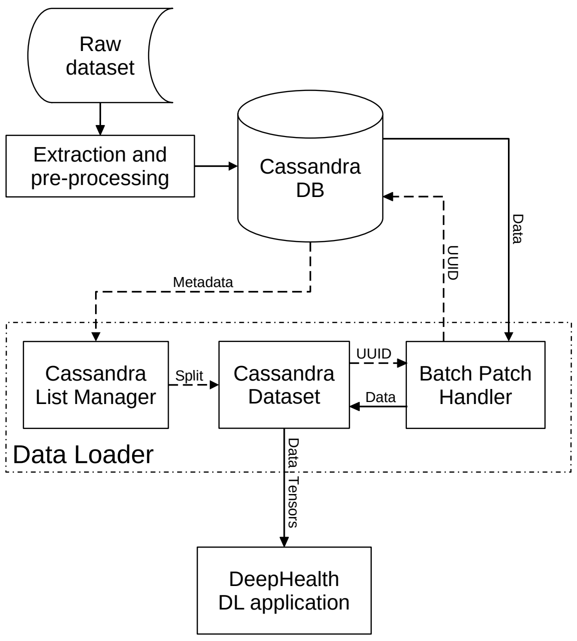


Figure 1: System architecture diagram of our data loader. Images are extracted from the raw dataset and pre-processed to be inserted, together with relevant metadata, in the Cassandra DB. The DB is subsequently queried to build the list of splits and to fetch images and labels whenever needed by the DL application.

The *metadata_by_nat* table is used when creating the splits, and allows to efficiently retrieve the full list of patients, slides and labels and to fetch the UUIDs of the patches for any given patient/slide/label combination. Once we have these data it is relatively easy[1] to ask the system, e.g., to create 5 splits, with different patients in each split, using a total of 1 million patches with size ratios $[5, 2, 1, 1, 1]$ among the 5 splits, and keep balanced labels (1:1 normal/tumor ratio).

The splits are expressed as lists of UUIDs and, once they have been created, they can easily be saved and loaded as needed. Subsequently, the training process will only need to access the *data_by_uuid* table, via efficient queries to single-row partitions. Note that more than one data table may be created, e.g., one might also want to save a color-normalized dataset, along with the original one.

Finally, the optional *metadata_by_uuid* table can be used at later stages of the DL workflow. E.g., when analyzing the training results one might want to trace misclassified patches back to the slides from which they have been extracted, to check for systematic errors in the original labeling.

4 Implementation

The data loader module is written in C++ and Python, and it is made up of three main classes (as shown in the system architecture diagram of Fig 1):

CassandraListManager This high-level Python class takes care of creating the splits, given the desired target parameters. Details in Sec. 4.1.

[1]The problem of optimally partitioning the dataset into splits is in general NP-hard. However, since in practice optimality is not required, some quick heuristics can be adopted to obtain an approximate solution [11].

BatchPatchHandler This low-level C++ class (with Python bindings exposed via pybind11 [23]) takes care of efficiently retrieving a batch of features and labels. It accepts in input a list of UUIDs and applies data-augmentation via the ECVL library [4], if needed. Details in Sec. 4.2.

CassandraDataset This is the main interface for using the data loader. It is written in Python and offers simple methods to load split files and fetch batches of data (features and labels). Its use is pretty straight-forward, as can be seen in the minimal example in Listing 1.

Listing 1 Example of data-loader use

```
from cassandra_dataset import CassandraDataset
from cassandra.auth import PlainTextAuthProvider

## Cassandra connection parameters
ap = PlainTextAuthProvider(username='user', password='pass')
cd = CassandraDataset(ap, ['cassandra-db'])

## Create (and save) list of splits
cd.init_listmanager(
  table='tissue_patches.metadata_by_nat', id_col='patch_id',
  partition_cols=['patient_id', 'slide_num', 'label'],
  split_ncols=1, num_classes=2)
cd.read_rows_from_db()
cd.init_datatable(table='tissue_patches.data_by_uuid')
cd.split_setup(split_ratios=[7,2,1], max_patches=1000000)
cd.save_splits('splits/1M_3splits.pckl')

## One-line alternative:
##   load an already existent list of splits
# cd.load_splits('splits/1M_3splits.pckl')

epochs = 50
split = 0 # training
cd.set_batchsize(32)
for _ in range(epochs):
    cd.rewind_splits(shuffle=True)
    for _ in range(cd.num_batches[split]):
        x,y = cd.load_batch(split)
        ## feed features and labels to DL engine [...]
```

4.1 CassandraListManager – Split creation

The automatic creation of splits works as follows:

- First, the list of Cassandra DB partitions is read.
- Then, the list of UUIDs contained in each partition is read and they are aggregated based on the chosen keys (e.g., patches are aggregated based on `patient_id`, so that patches of the same patient will all belong to the same split and thus will either be in the training or in the validation set).
- Each aggregated partition (group) is assigned to a split, so that the target values for each split and class are approximately met. (In more detail, the

desired target values are computed for each split/class combination and groups are assigned in round robin to the splits, provided they do not make them overflow. Finally, remaining groups are assigned to the splits randomly.)

- Once a bag of groups for each split has been computed, the rows are extracted in round robin from each group (to maximize diversity), until the target values are reached (or no more rows are available).

Note that given the standardized way in which data are stored in the DB and splits are formed (as lists of UUIDs), it is relatively easy to extend the split creation process with custom code, using the full list of rows which can be retrieved from the DB.

4.2 BatchPatchHandler – Performance optimizations

In order to increase the loader throughput, we have adopted the following optimizations in our code:

- Data for each split are read in parallel by a thread pool (with 32 threads as default).
- Data are prefetched in background, while the GPU is processing the previous (mini-)batch.
- Data augmentations are applied in background as well.
- Double-buffering is used to reduce the DB+network latency: i.e., the download of a second batch starts while the first one is still in progress, thus halving the average batch latency.
- Expensive system resources, such as threads and Cassandra connections, are allocated lazily. This means that only splits effectively being used do consume resources and it is thus possible to have many unused splits without impacting the system performance (see application in Sec. 6).

5 Evaluation and discussion

In this section we analyze our data management strategy, with the objective of identifying possible performance bottlenecks. In particular we want to measure the communication performance both on the client (data loader) and server (Cassandra) side. For the client side, we will measure the maximum throughput achievable by a single data loader, when it does not have to wait for computations on the retrieved data (i.e., cutting the actual GPU work). For the server side we will verify that our Cassandra servers are able to saturate their outgoing bandwidth, and that their retrieval time does not grow too much under heavy traffic load.

To allow better reproducibility we have tested our data loader using the standard ImageNet-2017 dataset (166 GB, 1,281,167 images, 1000 classes) [24]. Our test system is a cluster of 18 nodes, up to 2 running Cassandra DB and up to 16 consuming the data. The nodes are equipped with Intel Xeon E5-2680 v3 CPUs (12 cores, 2 threads/core) and are connected via a 10 Gb/s Ethernet (used by Cassandra) and a 56 Gb/s InfiniBand (used by our MPI parallel DL trainer for exchanging data, see Sec. 6). The nodes do not have GPUs, but since in this work we are only interested in the data loading stage, we have simply simulated their presence (by means of appropriate time sleeps) whenever needed.

For portability reasons we chose not to run on bare metal, but we have instead adopted Docker containerization with Kubernetes orchestration, both for Cassandra servers and for our data loader. The use of containers can introduce some network overheads, but these are mostly negligible and more than compensated for by the ease of deploying and managing the system [1].

5.1 Populating the DB

We have resized and center-cropped all the images to the standard resolution 224x224x3 (RGB) and saved them as BLOBs in the Cassandra DB, both as JPEG (quality: 90, average size: 20 kB) and not compressed TIFF (size: 150 kB). This data preprocessing step is easily parallelizable and scalable (no synchronizations are needed) and we implemented it with PySpark [31].

5.2 Performance of the data loader (client)

We have first tested the raw performance of a "short-circuited" data-loader, i.e., one which reads as many batches as possible, without actually consuming the data. Results are shown in Table 1 and Figure 2. For the smaller (JPEG) images the throughput is between 11,000 and 18,000 images per second, whereas for the not compressed TIFFs it is between 3800 and 4900 images per second, peaking when the batch size is 128. Since the thread parallelism is always 32, the smallest batch size pays the maximum latency out of 32 images, whereas, when the batch size increases, the latencies are averaged between subsequent rounds of retrieval, and hence the throughput increases (for example, if the batch size is 128, the retrieval time of each thread is the sum of 4 sequential transfers). However, as the batch size continues to grow, so does the stress on the Cassandra server while serving a batch, increasing the retrieval latency, which in turn decreases the overall throughput. This behavior is more evident in the case of uncompressed images as shown in Figure 2. In fact, assuming full bandwidth and a conservative network latency of 30 μs, transferring 20 KB and 150 KB on a 10 Gb Ethernet takes, respectively, less than 50 μs and 150 μs. Comparing these network transfer times with the DB retrieval times shown in Table 2, we can see how the latter tend to dominate the overall communication time.

As for the computational resources required in the data loading we note that they depend roughly on the transaction rate (assuming no data augmentation is performed). At maximum throughput, transferring 18k JPEG/s results in a CPU load of about 1900% (i.e., 19 threads at full speed, hence close to CPU saturation), whereas when moving 5k TIFF/s the load is about 400%. Considering that ResNet-50 [12], the standard network when testing the ImageNet dataset, consumes about 200 images/s on an NVIDIA TITAN RTX GPU, we can see that, depending on the chosen batch size, a single data loader can sustain about 50-90 GPUs, when transferring compressed JPEG, and 19-24 when using not compressed TIFF.

Table 1: Single data-loader performance (retrieving data from a single Cassandra server).

Batch size	**Image size** [kB]	**Time per batch** [ms]	**Throughput** [img/s]	[MB/s]
32	20	2.80 ± 0.018	11440 ± 70	230
64	20	4.87 ± 0.017	13150 ± 45	260
128	20	7.58 ± 0.024	16900 ± 55	340
256	20	13.9 ± 0.065	18400 ± 85	370
512	20	27.9 ± 0.15	18340 ± 95	370
32	150	7.07 ± 0.076	4530 ± 50	680
64	150	13.2 ± 0.13	4830 ± 45	720
128	150	26.3 ± 0.40	4870 ± 70	730
256	150	61.3 ± 1.3	4180 ± 90	630
512	150	132 ± 1.9	3870 ± 60	580

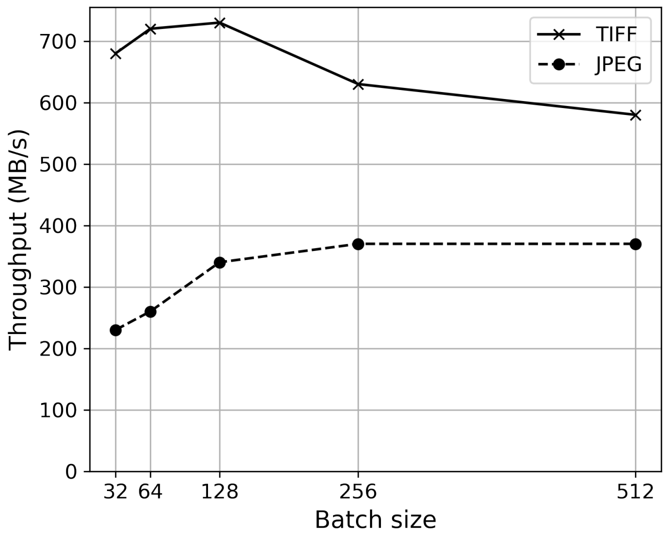


Figure 2: Throughput (MB/s) of the “short-circuited” data-loader (reading as many batches as possible) at different batch sizes. The graph presents two plots, showing the data rate for compressed (JPEG, dashed line) and not compressed (TIFF, solid line) images. As explained in 5.2, the not compressed case exhibits some performance degradation as the batch size increases.

5.3 Performance of Cassandra DB (server)

In this section we investigate the behavior of Cassandra server nodes under heavy load. We are interested in particular in seeing whether they can saturate the outgoing bandwidth (10 Gb/s) when flooded by data requests, and if they can service these requests while keeping the DB latency stable. To this purpose we have measured the distribution of Cassandra retrieval latency (via the `nodetool tablehistograms` command) both with 1 and 16 active, short-circuited data loaders (with batch size = 256) which try to read as many data as possible from the servers. The results, in Table 2, show that the network can be saturated both when retrieving compressed and not compressed images and that read latencies up to the 95th percentile remain almost constant even when the network is saturated, whereas the 99th percentile grows approximately by a factor 2. As for the computational intensity, on a heavy-loaded Cassandra node we measure 1000% CPU usage when retrieving JPEGs and 700% with TIFFs.

The image rates at saturation are about 50,000 images/s for 20 kB JPEGs and 7000 images/s for not compressed 150 kB TIFFs, which amounts to serving enough data to feed, respectively, 250 and 35 GPUs, per Cassandra node.

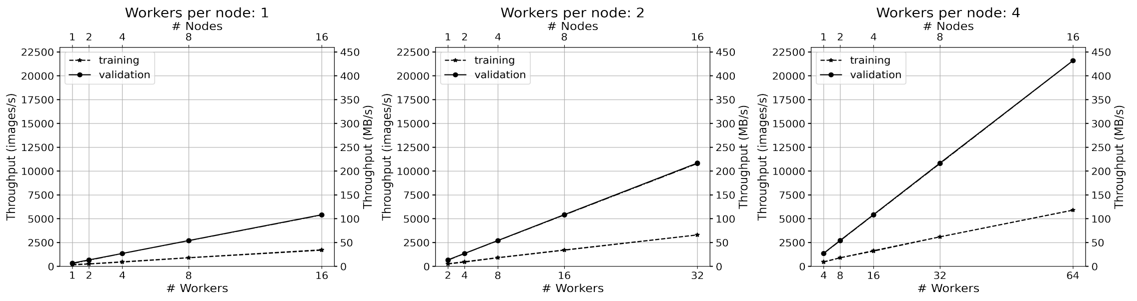
5.4 Scaling up/down Cassandra DB

Cassandra DB allows for nodes to be added/removed to/from an existing ring (i.e., a Cassandra cluster), without any service disruption. We have verified that, when activating a second Cassandra node under heavy load, the outgoing bandwidth on both nodes is still saturated, and that when the second node is subsequently deactivated the load on the first one remains stable. Note that the scaling up/down of the ring has been performed while 16 loaders were continuously retrieving data from the DB, without any service interruption.

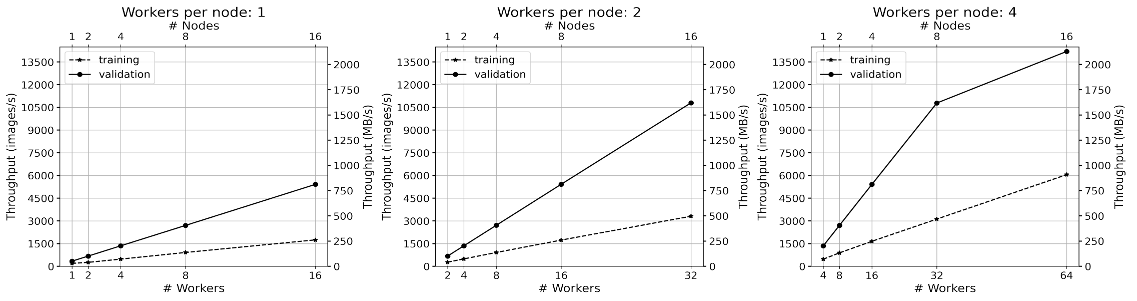
5.5 Discussion

5.5.1 Performance comparison with parallel filesystems

Our data loader, compared to high-end parallel filesystems, has a major performance disadvantage: communications to Cassandra servers are TCP based, whereas in parallel filesystems there can be RDMA transfers directly from the storage to the consuming nodes (if the network supports them) and this impacts both network latency and CPU usage. However, since the retrieval latency dominates the network one, the performance gap is not too wide: for example, a parallel filesystem installation using BeeGFS has latency in the order of 100 μs and can support up to 250,000 operations/s per node [3]. In our case, we can fetch 150 KB images with latency of about one millisecond and we can reach bandwidth saturation at rate of 50,000 transfers/s per server node for 20 KB images, using a 10 GbE network and general purpose nodes. Parallel filesystem, on the other side, do not simplify the management of splits and metadata, as our approach does. Overall, we think that our design can be of particular interest for small and medium size systems, showing a good trade-off among performance, cost and ease of deployment.



(a) Compressed images



(b) Not compressed images

Figure 3: Throughput of the Distributed Deep Learning use case. The rows represent the results, using compressed (JPEG) and not compressed (TIFF) images. The columns show the different behavior as the number of worker per node increases from 1 (on the left most column) to 4. For all the graphs the horizontal and vertical axis are duplicated to show complementary information. The horizontal axes show respectively the number of actual workers used by the application (bottom) and the number of active nodes the workers are running on (top). The vertical axes show two different measures of throughput: the number of images per second processed (left axis) and the data rate in MB per second (right axis). Each graph presents two plots, showing the training and the validation tasks.

Table 2: Cassandra read latencies at medium load and at network saturation (10 Gb/s Ethernet). Batch size = 256. The leftmost tables refer to 1 data loader, whereas the rightmost ones to 16 data loaders.

Percentile [%]	**Read latency** [μs]
50	88.15
75	105.78
95	152.32
98	182.79
99	263.21

(a) JPEG, 370 MB/s

Percentile [%]	**Read latency** [μs]
50	88.15
75	105.78
95	152.32
98	263.21
99	454.83

(b) JPEG, **1060** MB/s

Percentile [%]	**Read latency** [μs]
50	454.83
75	545.79
95	943.13
98	1629.72
99	1955.67

(c) TIFF, 630 MB/s

Percentile [%]	**Read latency** [μs]
50	454.83
75	545.79
95	1131.75
98	1955.67
99	4055.27

(d) TIFF, **1060** MB/s

5.5.2 Floating point vs integer data

Our use case utilizes integer data (i.e., RGB images), but in many scientific application this does not happen: for example one might apply DL techniques to general tensors, where each spatial point is associated with floating point data (e.g., 64-bit), and, if the data are not compressed, an 8x increase in the required bandwidth has to be taken into account (compared to our not compressed TIFF case). This means that a 10 GbE can sustain only up to 4 GPUs. One approach that could be explored to help scaling the computation in this case (apart from using a faster network connection) would be using each computing node also as a Cassandra server, somehow resembling the solution adopted in [15] for synthetic data.

6 Distributed Deep Learning

An application that can benefit hugely from our data loading solution is distributed DL [2]. In this section, after a brief introduction to parallelization methods for DL, we present a simple distributed training that leverages our data management strategy and we analyze its implementation and performance.

6.1 Data and model parallelism

The two main approaches to parallelize DL algorithms are data and model parallelization [2].

A general workflow for data parallelism is the following:

- the neural network (NN) to be trained is copied to all the computational devices (e.g., GPUs);
- at every iteration, the current (global) mini-batch of samples is divided into chunks (local mini-batches), that are mapped to the local copies of the NN;
- at the end of each iteration the local gradients, computed after a back-propagation pass, are aggregated among all (or almost all) the workers, to compute the global gradient and update the network parameters.

In a distributed system the average of the gradients is typically implemented with an All-Reduce operation, followed by a local update of the parameters. This synchronization step affects the parallel efficiency of the distributed training, thus limiting the scalability of the computation. Another, subtler, scalability issue is related to the mini-batch size: as the parallelism increases, so does the global batch size and this can affect the generalization capability of the model [2].

In model parallelism the computations of different parts of a neural network are performed by different devices (e.g., GPUs). In this case the mini-batch size is independent of the parallelism, and hence there is no reduction of generalization capabilities as in data parallelism. On the other hand, the main drawback of this approach is given by the high communication costs due to the dependencies among different parts of the NN. Some enhanced architectures have been proposed in literature to mitigate this overhead by using redundant computations, however model parallelism is typically used when the NN cannot fit on a single computational device or if the particular NN architecture (e.g., LSTM models) can be efficiently split across different devices.

6.2 Decentralized data distribution

In order to stress our data loading pipeline, we have implemented a basic version of synchronized data parallelism for the EDDL library [4], extending the SGD optimizer by using Open MPI to compute the average of gradients, the losses and the performance metrics among all the parallel ranks. Note that this approach applies transparently to both inter- and intra-node communications.

We chose to keep a copy of synchronized parameters on each worker (an approach also called *mirrored strategy* [9]), instead of using a centralized parameters server, as we are not interested on implementing more complex distributed schemas like asynchronous updates. We also chose to update NN parameters at each iteration to closely mimic the behavior of the original SGD algorithm.

Our data management strategy allows to easily distribute (and uniformly, globally permute) data among the MPI ranks, without the need for centralized process, as it is exemplified by the following procedure for a parallel system of size n:

- At startup, each rank reads the full list of the images hosted by the Cassandra servers. This can be done either by querying directly the DB or by reading a pre-shared file (of size about 60 MB for ImageNet).
- The data loader and the network on each rank are initialized with the same seed (e.g., broadcasted by rank 0).
- Each data loader creates $2n$ splits: n for training and n for validation (as described in § 4.1).

- Rank i will read training data from split i and validation data from split $n + i$.
- At the end of an epoch the UUIDs in the training splits are shuffled (again, using the same seed on each rank) and the next epoch can start.

Some observations:

- In EDDL is currently impossible to set the seed for the network initialization, hence at startup we broadcast the network parameters from rank 0, to be able to start the training everywhere in the same state.
- Since resources consumed by splits are lazily allocated, the load on each rank remains constant when the parallelism grows.

6.3 Simulation of multi-GPU training

We have adapted our MPI distributed learner to simulate the load on the Cassandra servers induced by different training configuration (up to 16 nodes, up to 4 GPUs per node) in the following way:

- The GPU computations have been replaced by appropriate time sleeps, obtained by actual performance measures on an NVIDIA TITAN RTX GPU.
- The communications are normally carried out, using the UCX module [29] (in our system: inter-node via InfiniBand, intra-node via shared memory).

We have chosen to assign 6 threads per MPI rank (i.e., options `--map-by node:pe=6 --bind-to core` of `mpirun`) and we have specified in the `hostfile` a number of slots per each node equal to the number of simulated GPUs.

After the initial setup phase, in which the splits are created, each worker starts the loop across the epochs. During each epoch two inner loops across the local batches are performed, respectively to train and validate the model. From the point of view of the data loader the operations performed are identical, because the only difference is the split index used to get the local batch. However, the operations simulated on the retrieved data are different. The training loop involves a local forward and backward propagation (simulated by time sleeps), followed by the gradient average operation (which is instead performed in full, exactly as in the case where GPUs are available), run to keep the network copies synchronized at the end of every iteration. The validation loop, on the other hand, computes only a local forward pass (simulated) followed by a global average for losses and metrics (fully performed). Since these average operations involve only floating point communications, their overhead is negligible and the validation task behaves as an embarrassing parallel algorithm and scales linearly (up to saturation of the outgoing bandwidth of the Cassandra servers). Accordingly to the difference in the operations, the sleeps for training (forward and backward) and validation (only forward) are different, so as to match the actual values we have measured on GPUs.

The throughputs obtained by simulating distributed trainings with compressed and not compressed images are shown in Tables 3 and 4, and Figures 3a and 3b. As can be seen by the data, the validation measurements show a linear relationship between the number of running workers and the throughput. There is an exception for the validation of not compressed images with 64 workers,

Table 3: Data loading during distributed training. Compressed files (JPEG, 20 kB), batch size = 28.

# Nodes	Node load	Training [img/s]	Training [MB/s]	Validation [img/s]	Validation [MB/s]
1	1	194	3	340	6
2	1	263	5	677	13
4	1	479	9	1353	27
8	1	912	18	2710	54
16	1	1739	34	5420	108
1	2	279	5	681	13
2	2	487	9	1356	27
4	2	918	18	2713	54
8	2	1723	34	5428	108
16	2	3303	66	10848	216
1	4	478	9	1363	27
2	4	902	18	2725	54
4	4	1648	32	5422	108
8	4	3101	62	10841	216
16	4	5908	118	21602	432

where the throughput saturates the bandwidth of the 2 Cassandra servers being used.

7 Software availability

Our software is free, distributed under the MIT License. The code used in this article is available in the branch *distributed* of the following GitHub repository: `https://github.com/deephealthproject/promort_pipeline`. Up-to-date code, as well as future updates, can be found here: `https://github.com/deephealthproject/CassandraDL`.

8 Conclusion and future work

In this work we have introduced a novel strategy to efficiently and scalably manage data in DL processes by leveraging Apache Cassandra NoSQL DB, and we have shown the ease of use and performance of our newly developed data loader in the context of image classification. In the future we would like to extend our data loader to include more DL frameworks (e.g., TensorFlow or PyTorch) and extend its functionalities to include, e.g., support for image segmentation problems.

Acknowledgments

This work has been funded by the European Commission under the H2020 program grant DeepHealth (n. 825111). We would like to thank NVIDIA for the

Table 4: Data loading during distributed training. Not compressed files (TIFF, 150 kB), batch size = 28. Throughput during the validation phase in the last row ($16 \cdot 4 = 64$ MPI ranks) saturates the combined bandwidth (20 Gb/s) of the 2 Cassandra servers in our ring.

# Nodes	**Node load**	**Training** [img/s]	**Training** [MB/s]	**Validation** [img/s]	**Validation** [MB/s]
1	1	195	29	340	51
2	1	263	39	677	101
4	1	478	71	1354	203
8	1	916	137	2705	405
16	1	1747	262	5414	812
1	2	274	41	677	101
2	2	492	73	1355	203
4	2	913	137	2712	406
8	2	1733	260	5421	813
16	2	3310	496	10802	1620
1	4	472	70	1350	202
2	4	891	133	2706	405
4	4	1655	248	5415	812
8	4	3123	468	10788	1618
16	4	6059	908	14198	2129 *

donation of a TITAN RTX GPU (as part of the CADL 2020 BPA), that we have used in our tests. Finally, we thank E. Gobbetti and L. Pireddu for their insightful advice while reviewing the draft, and our system administrators M. Moro and M. Muggiri, for their continuous support during the evaluation phases.

References

[1] Angel M Beltre et al. “Enabling HPC workloads on cloud infrastructure using Kubernetes container orchestration mechanisms”. In: *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. IEEE. 2019, pp. 11–20.

[2] Tal Ben-Nun and Torsten Hoefler. “Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis”. In: *ACM Comput. Surv.* 52.4 (Aug. 2019). ISSN: 0360-0300. DOI: `10.1145/3320060`. URL: `https://doi.org/10.1145/3320060`.

[3] Peter Braam. *Shaping tomorrow with BeeGFS. The architecture, technology, product direction and development plans.* ISC High Performance 2021. 2021.

[4] Michele Cancilla et al. “The DeepHealth Toolkit: a unified framework to boost biomedical applications”. In: *2020 25th International Conference on Pattern Recognition (ICPR)*. IEEE. 2021, pp. 9881–9888.

[5] Philip Carns et al. “Small-file access in parallel file systems”. In: *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE. 2009, pp. 1–11.

[6] *Cassandra Case Studies*. https://cassandra.apache.org/case-studies/. 2021.

[7] Ali Davoudian, Liu Chen, and Mengchi Liu. "A survey on NoSQL stores". In: *ACM Computing Surveys (CSUR)* 51.2 (2018), pp. 1–43.

[8] Nikoli Dryden et al. "Clairvoyant prefetching for distributed machine learning I/O". In: *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri, USA, November 14 - 19, 2021*. Ed. by Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin. ACM, 2021, 92:1–92:15. DOI: 10.1145/3458817.3476181. URL: https://doi.org/10.1145/3458817.3476181.

[9] Aurelien Geron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. 2nd. O'Reilly Media, Inc., 2019. ISBN: 1492032646.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[11] Brian Hayes. "Computing science: The easiest hard problem". In: *American Scientist* 90.2 (2002), pp. 113–117.

[12] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[13] Marcin Krotkiewski. *Dealing with small files in HPC environments: automatic loop-back mounting of disk images*. 2017.

[14] Thorsten Kurth et al. "Exascale deep learning for climate analytics". In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2018, pp. 649–660.

[15] Nouamane Laanait et al. "Exascale deep learning for scientific inverse problems". In: *arXiv preprint arXiv:1909.11150* (2019).

[16] Paras Lakhani et al. "Hello world deep learning in medical imaging". In: *Journal of digital imaging* 31.3 (2018), pp. 283–289.

[17] Avinash Lakshman and Prashant Malik. "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[18] Paul J. Leach, Rich Salz, and Michael H. Mealling. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122. July 2005. DOI: 10.17487/RFC4122. URL: https://rfc-editor.org/rfc/rfc4122.txt.

[19] Xiuqiao Li et al. "Small files problem in parallel file system". In: *2011 International Conference on Network Computing and Information Security*. Vol. 2. IEEE. 2011, pp. 227–232.

[20] Geert Litjens et al. "1399 H&E-stained sentinel lymph node sections of breast cancer patients: the CAMELYON dataset". In: *GigaScience* 7.6 (2018), giy065.

[21] Pierre Matri et al. "TýrFS: increasing small files access performance with dynamic metadata replication". In: *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE. 2018, pp. 452–461.

[22] *ml-pyxis – Tool for reading and writing datasets of tensors (numpy.ndarray) with MessagePack and Lightning Memory-Mapped Database (LMDB)*. `https://github.com/vicolab/ml-pyxis`. 2021.

[23] *pybind11 — Seamless operability between C++11 and Python*. `https://github.com/pybind/pybind11`. 2021.

[24] Olga Russakovsky et al. "Imagenet large scale visual recognition challenge". In: *International journal of computer vision* 115.3 (2015), pp. 211–252.

[25] Pol Santamaria et al. "Evaluating the benefits of key-value databases for scientific applications". In: *International Conference on Computational Science*. Springer. 2019, pp. 412–426.

[26] Alexandru D Sicoe et al. "A persistent back-end for the ATLAS TDAQ online information service (P-BEAST)". In: *Journal of Physics: Conference Series* 368 (June 2012), p. 012002. DOI: `10.1088/1742-6596/368/1/012002`. URL: `https://doi.org/10.1088/1742-6596/368/1/012002`.

[27] Peter Ström et al. "Pathologist-Level Grading of Prostate Biopsies with Artificial Intelligence". In: *CoRR* abs/1907.01368 (2019). arXiv: `1907.01368`. URL: `http://arxiv.org/abs/1907.01368`.

[28] *TFRecord and tf.train.Example*. `https://www.tensorflow.org/tutorials/load_data/tfrecord`. 2021.

[29] *Unified Communication X — an open-source, production-grade communication framework for data-centric and high-performance applications*. `https://openucx.org/`. 2021.

[30] Chih-Chieh Yang and Guojing Cong. "Accelerating data loading in deep neural network training". In: *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE. 2019, pp. 235–245.

[31] Matei Zaharia et al. "Apache spark: a unified engine for big data processing". In: *Communications of the ACM* 59.11 (2016), pp. 56–65.

[32] Zhao Zhang et al. "Aggregating local storage for scalable deep learning i/o". In: *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*. IEEE. 2019, pp. 69–75.

[33] Yue Zhu et al. "Entropy-aware I/O pipelining for large-scale deep learning on HPC systems". In: *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE. 2018, pp. 145–156.