# Deep Learning Optimized on Jean Zay

## Dataset optimization

Storage spaces and data format

CNRS

IDRIS

# Dataset optimization

**Main bottlenecks** ◄

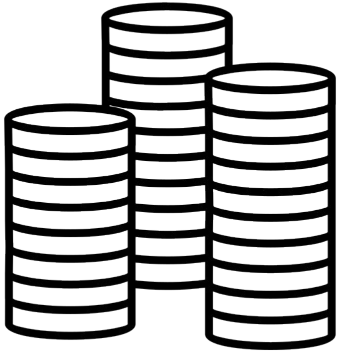
Data storage – various disk spaces ◄

Data format – at sample level ◄

Data format – at dataset level ◄

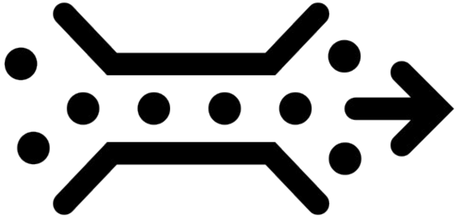
# Bottlenecks upstream of DataLoader

**Storage Disks**



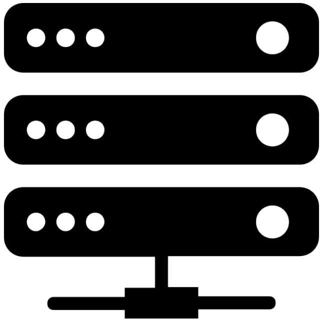
1. I/O performance

**Interconnection Network**
Omnipath



2. Shared Bandwidth

**CPU workers**



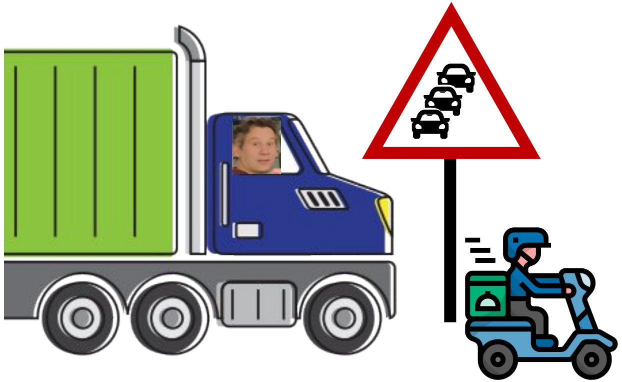
3. Decoder performance

# Bottlenecks upstream of DataLoader

**Storage Disks**

1. I/O performance

**Interconnection Network**
Omnipath



2. Shared Bandwidth

**CPU workers**

3. Decoder performance

# Dataset optimization

Main bottlenecks ◀

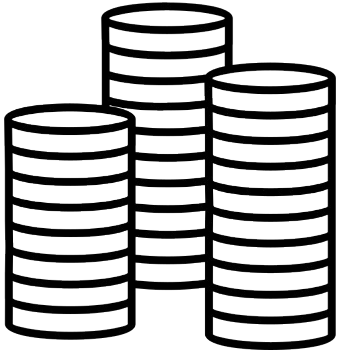
**Data storage – various disk spaces** ◀

Data format – at sample level ◀

Data format – at dataset level ◀

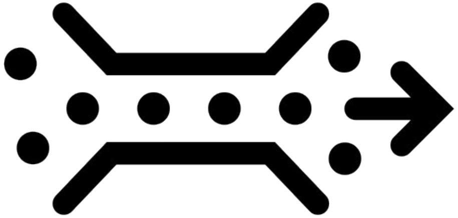
# Bottlenecks upstream of DataLoader

**Storage Disks**



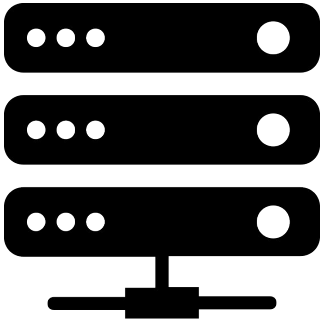
1. I/O performance

**Interconnection Network**
Omnipath



2. Shared Bandwidth

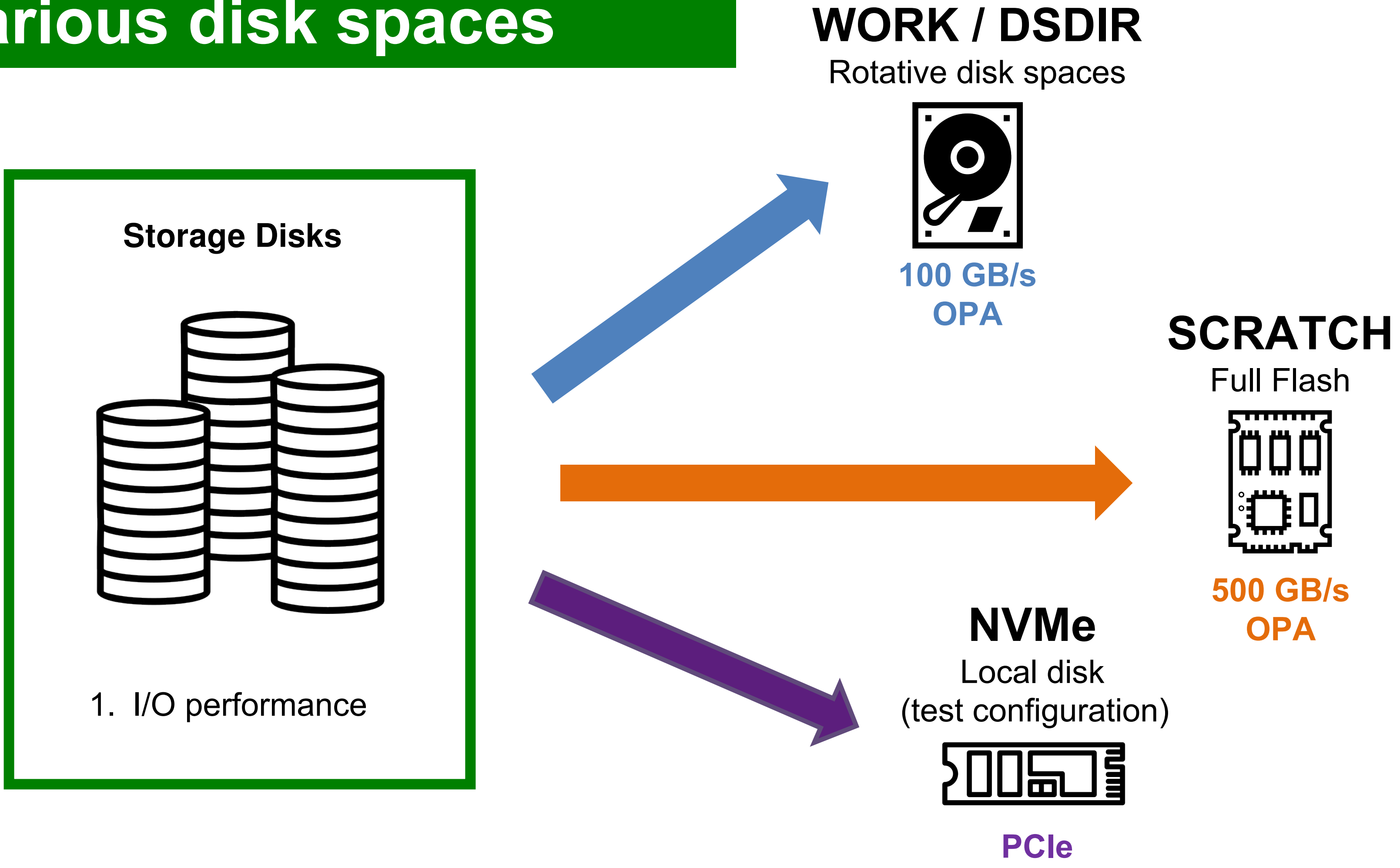
**CPU workers**



3. Decoder performance

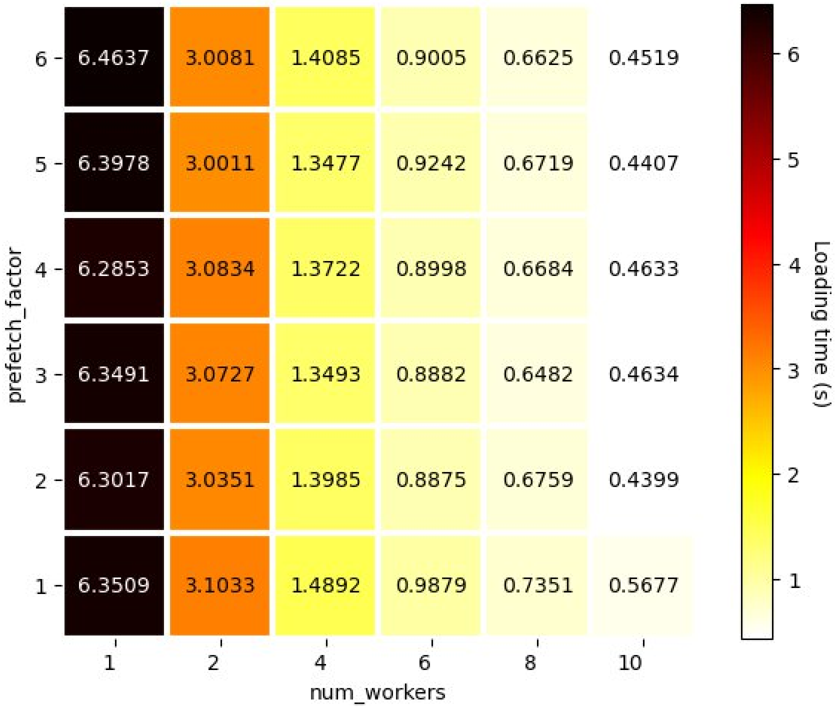
**Where should I store my dataset?**

# Various disk spaces



# Various disk spaces

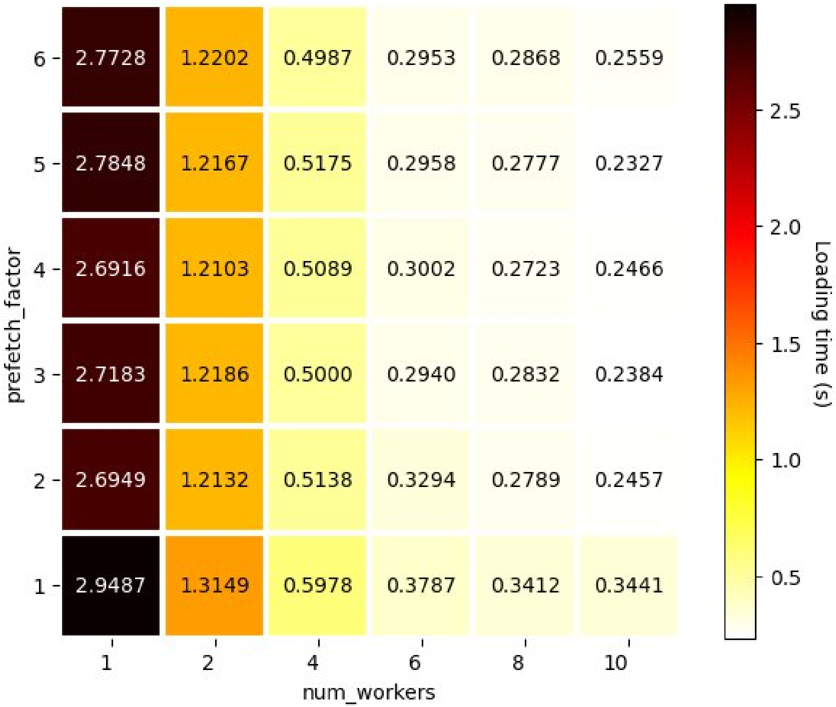
dlojz.py - 50 iterations - test partition gpu_p4




WORK / DSDIR

100 GB/s - OPA

÷ 2

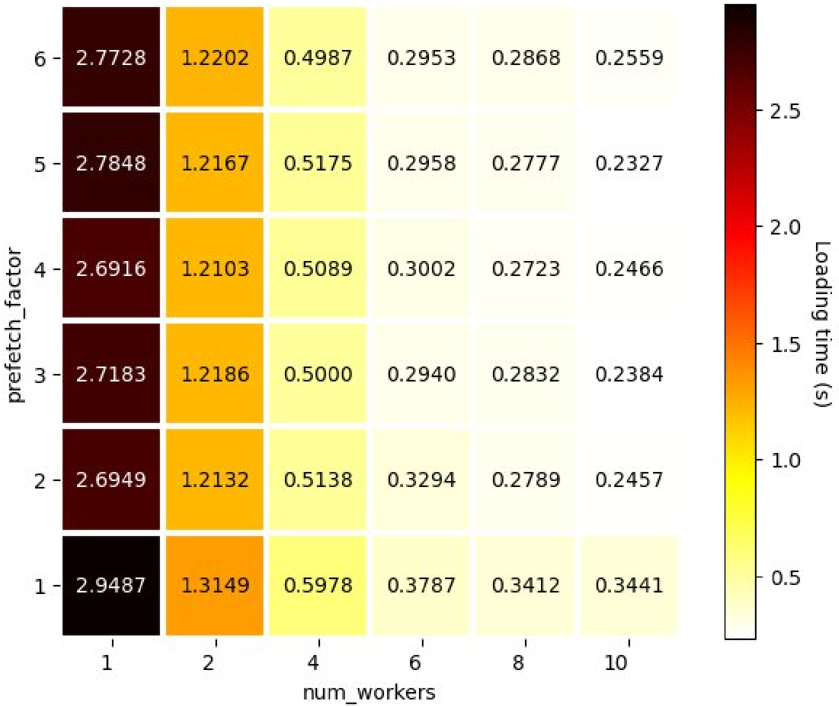


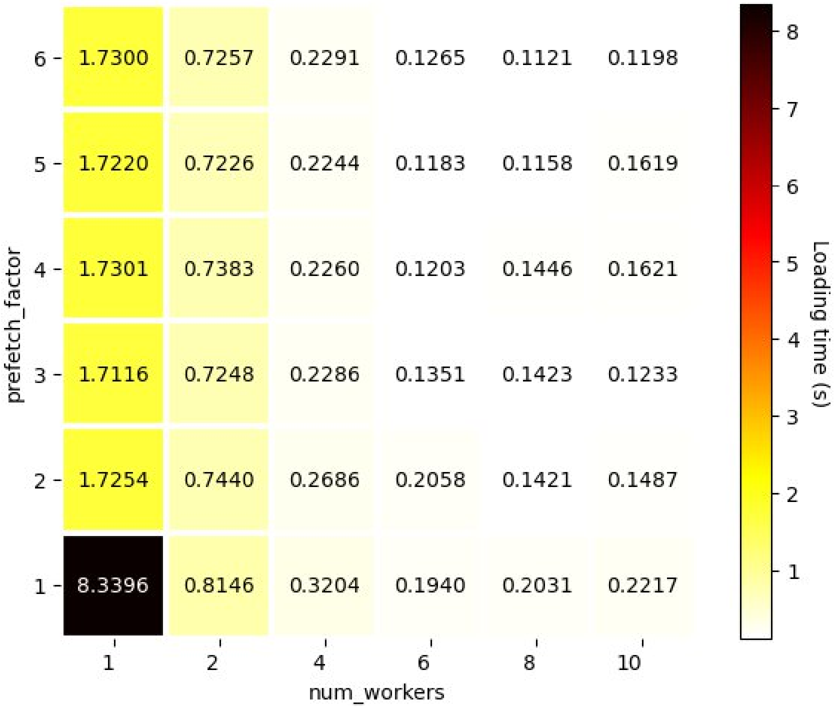

SCRATCH

500 GB/s - OPA

# Various disk spaces

dlojz.py - 50 iterations - test partition gpu_p4




SCRATCH

**500 GB/s - OPA**

÷ 2

NVMe

**PCIe**

# Various disk spaces

- **NVMe**
  - ✓ Best IO performance
  - – You need to copy your dataset on the local disk first, which can take a very long time
  - – This solution is not suitable at the scale of a supercomputer so it is not available to users

- **SCRATCH**
  - ✓ Second best IO performance
  - ✓ Very large quota (bytes and inodes)
  - – 30 days file lifespan
  - – Not backed up

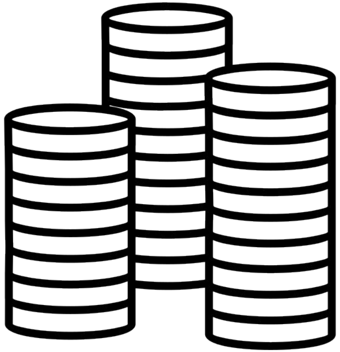
- **WORK / DSDIR**
  - – Worst performance (but it is still acceptable)
  - – Only 5 TB and 500k inodes
  - ✓ IDRIS support team manages the dataset for you in the DSDIR (downloading, preprocessing,...)
  - ✓ Backed up

# Dataset optimization

- Main bottlenecks ◄
- Data storage – various disk spaces ◄
- **Data format – at sample level** ◄
- Data format – at dataset level ◄

# Bottlenecks upstream of DataLoader

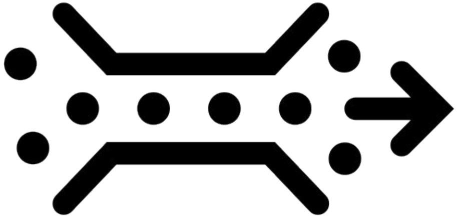
**Storage Disks**



1. I/O performance

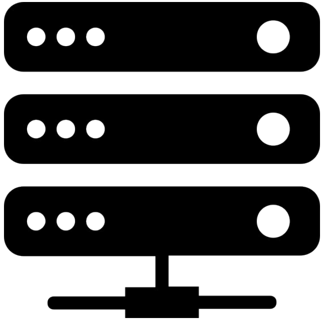
**Interconnection Network**

Omnipath



2. Shared Bandwidth

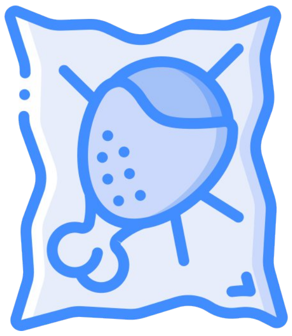
**CPU workers**



3. Decoder performance

**Which format for my data?**

# At sample level - Sample decoding



**Binary format**: Pickle format, hdf5,...
Decoded more quickly, takes more space

- ✓ Decoder performance
- – Shared bandwidth
- – Storage volume

**Compressed format**: jpeg, png,...
Decoded more slowly, takes less space

- – Decoder performance
- ✓ Shared bandwidth
- ✓ Storage volume

# Dataset optimisation

Main bottlenecks ◄

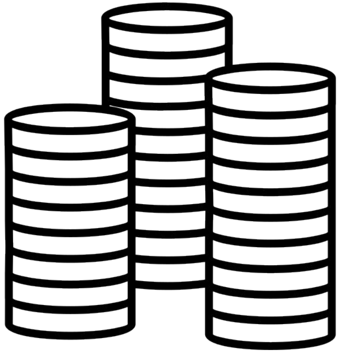
Data storage – various disk spaces ◄

Data format – at sample level ◄

**Data format – at dataset level ◄**

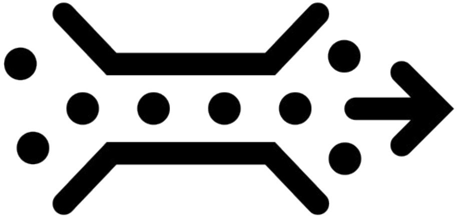
# Bottlenecks upstream of DataLoader

**Storage Disks**



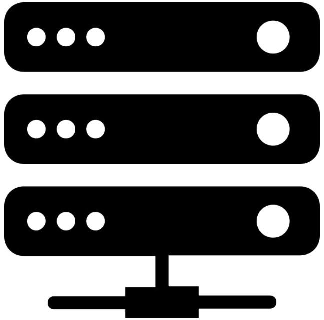
1. I/O performance

**Interconnection Network**
Omnipath



2. Shared Bandwidth

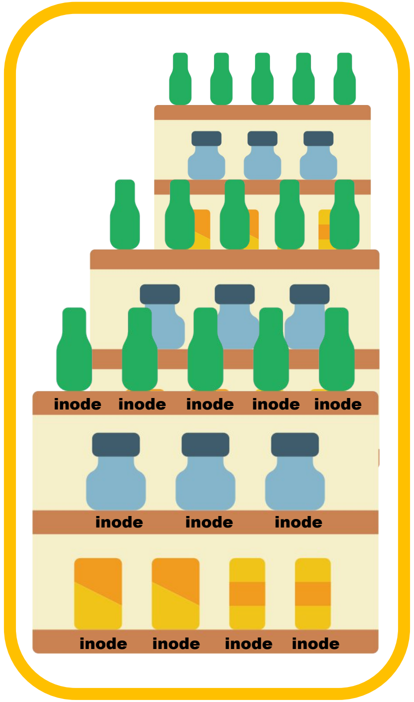
**CPU workers**



3. Decoder performance

**Which format for my dataset?**

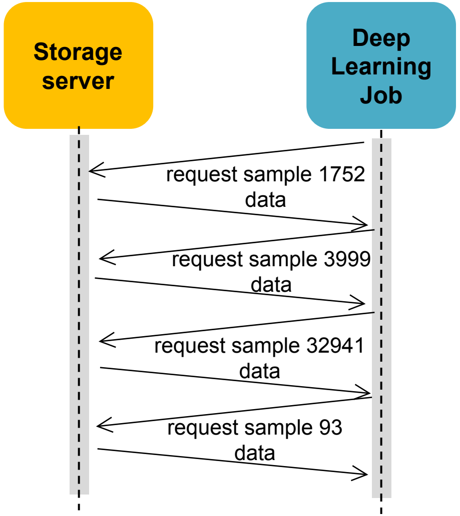
# Intuitive way

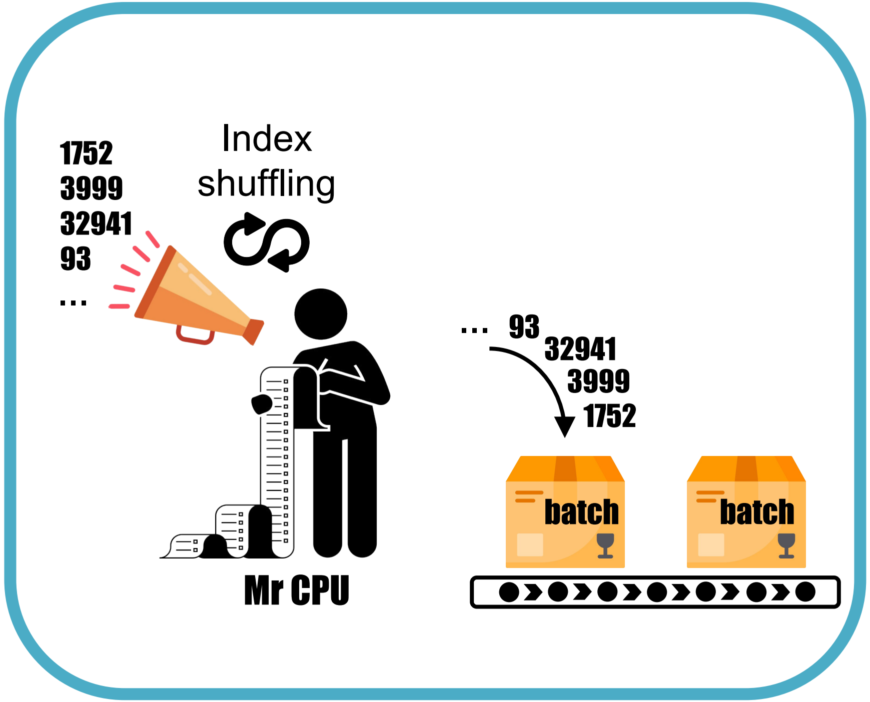



**Map-style dataset**

__getitem__

Random Access
to File Store




Pros: Easy to handle, random access possible

Cons: Lots of inodes, lots of I/Os

# Too many inodes is an issue

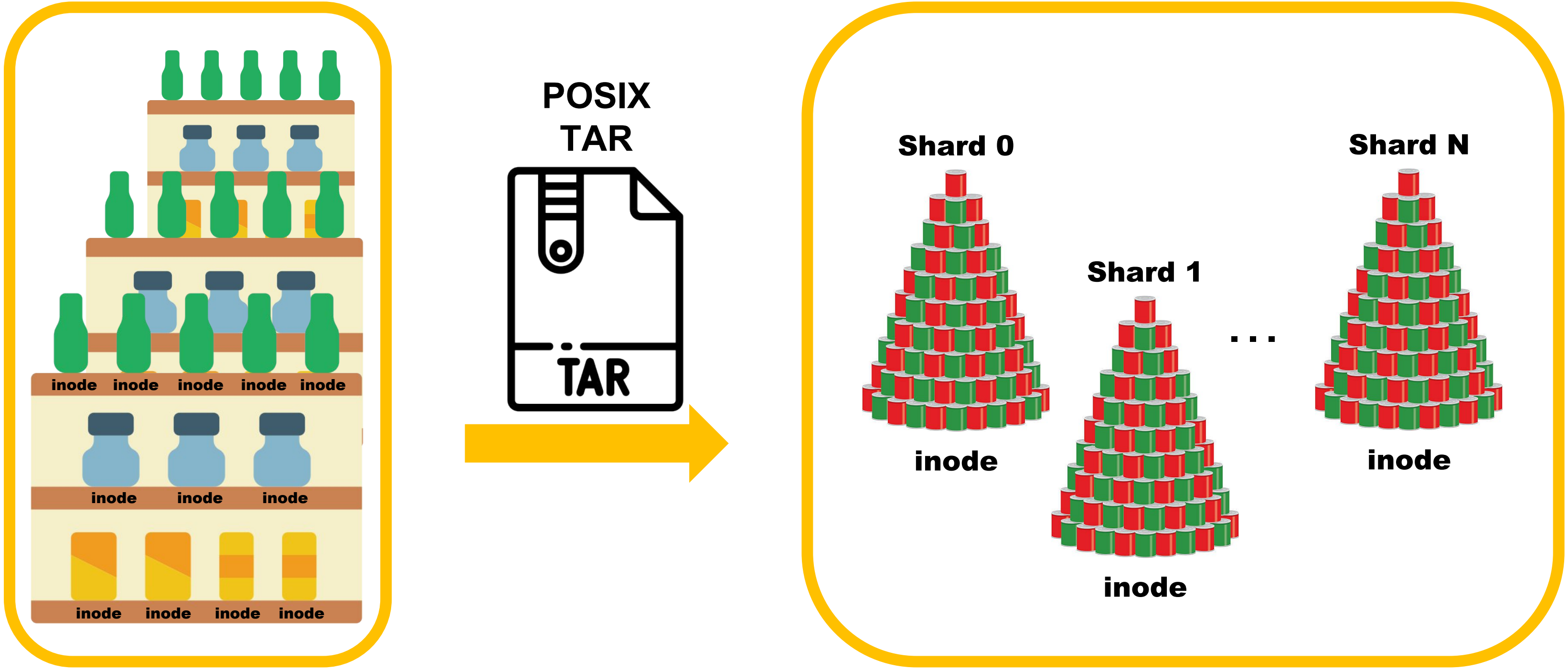
**Error: Disk quota exceeded**

Reminder:

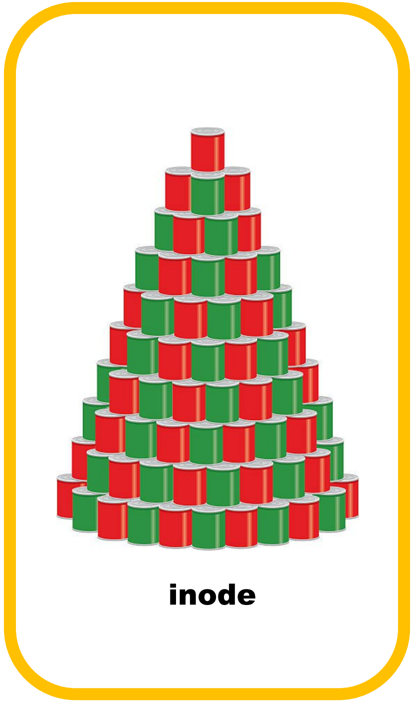
- $WORK quota per user is 5 TB / **500 kinodes**
- $SCRATCH safety quota per user is 250TB / **150 Minodes**

+ IBM Spectrum Scale file system does not like small file I/O intensive workloads

# WebDataset format – Gathering inodes



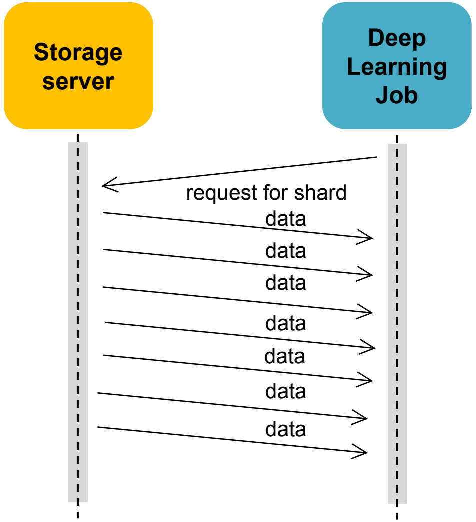
# WebDataset format – Iterable dataset

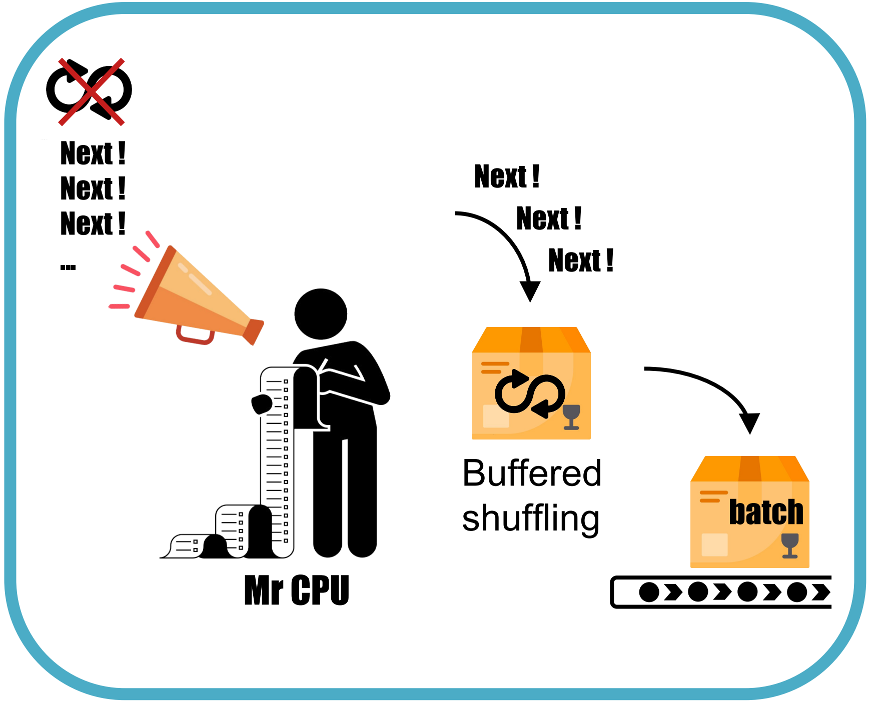



Iterable-style dataset

__iter__

Pipelined Access
to Object Store



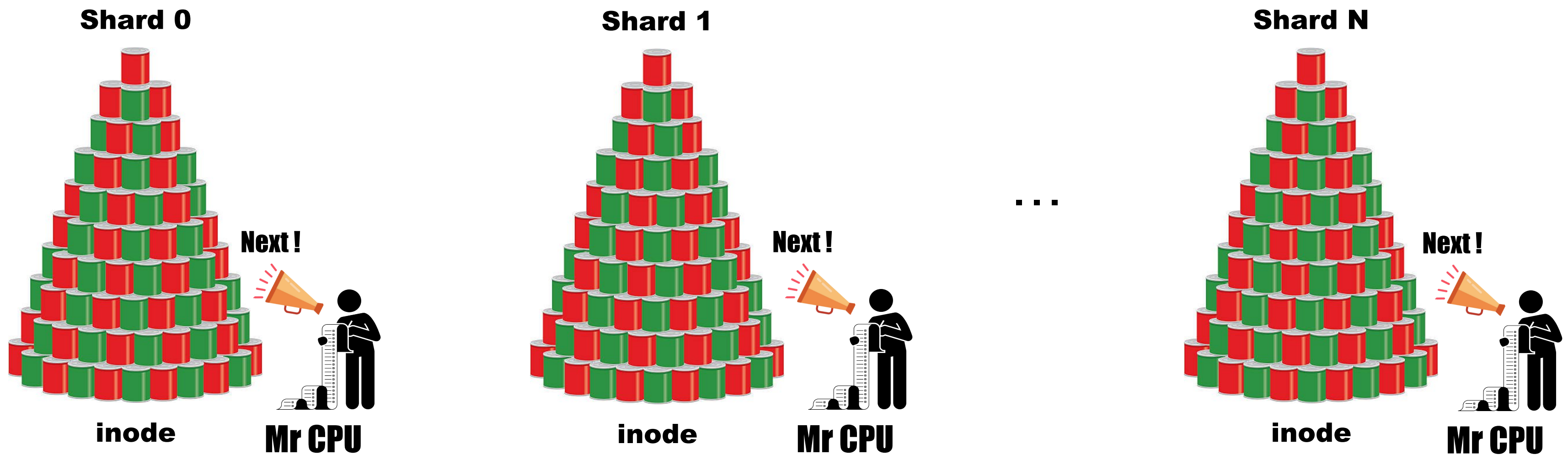




Pros: Fewer I/Os, fewer inodes

Cons: Difficult to shuffle or distribute, unknown dataset length

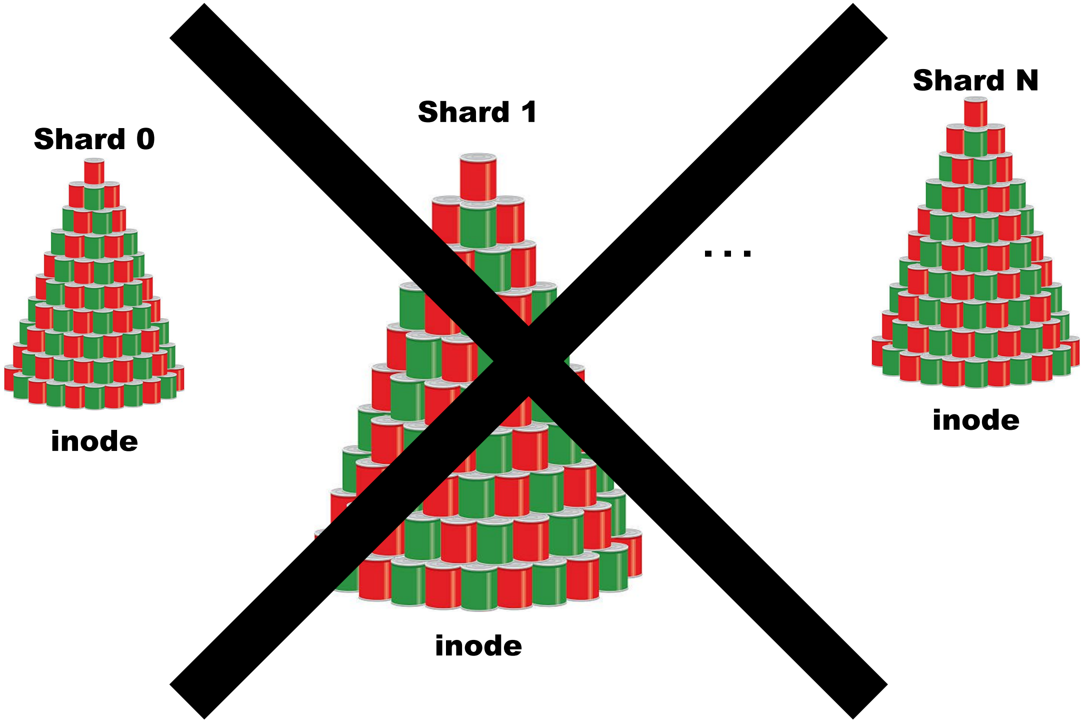
# WebDataset format - Sharding

Sharding is necessary to benefit from parallel implementation (DataLoader multi-processing and Distributed Data Parallelism).



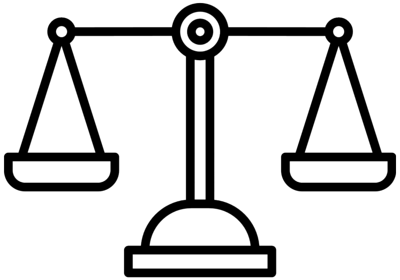

The number of shards should be a multiple of the number of tasks/GPUs.

# WebDataset format - Sharding

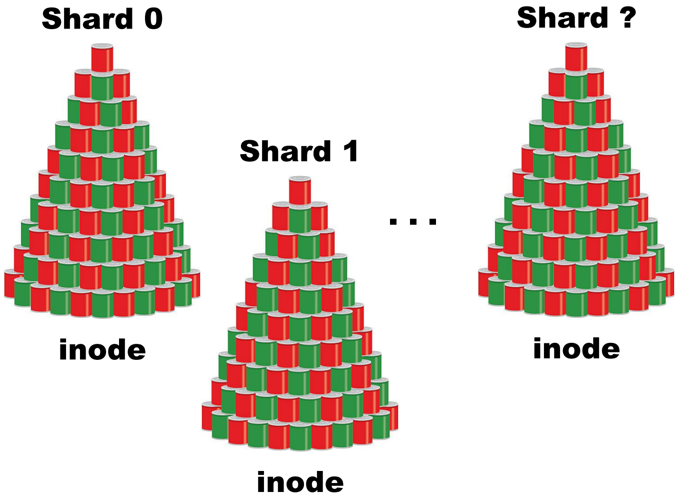



Samples must be evenly distributed among the shards to balance the workload between processes.

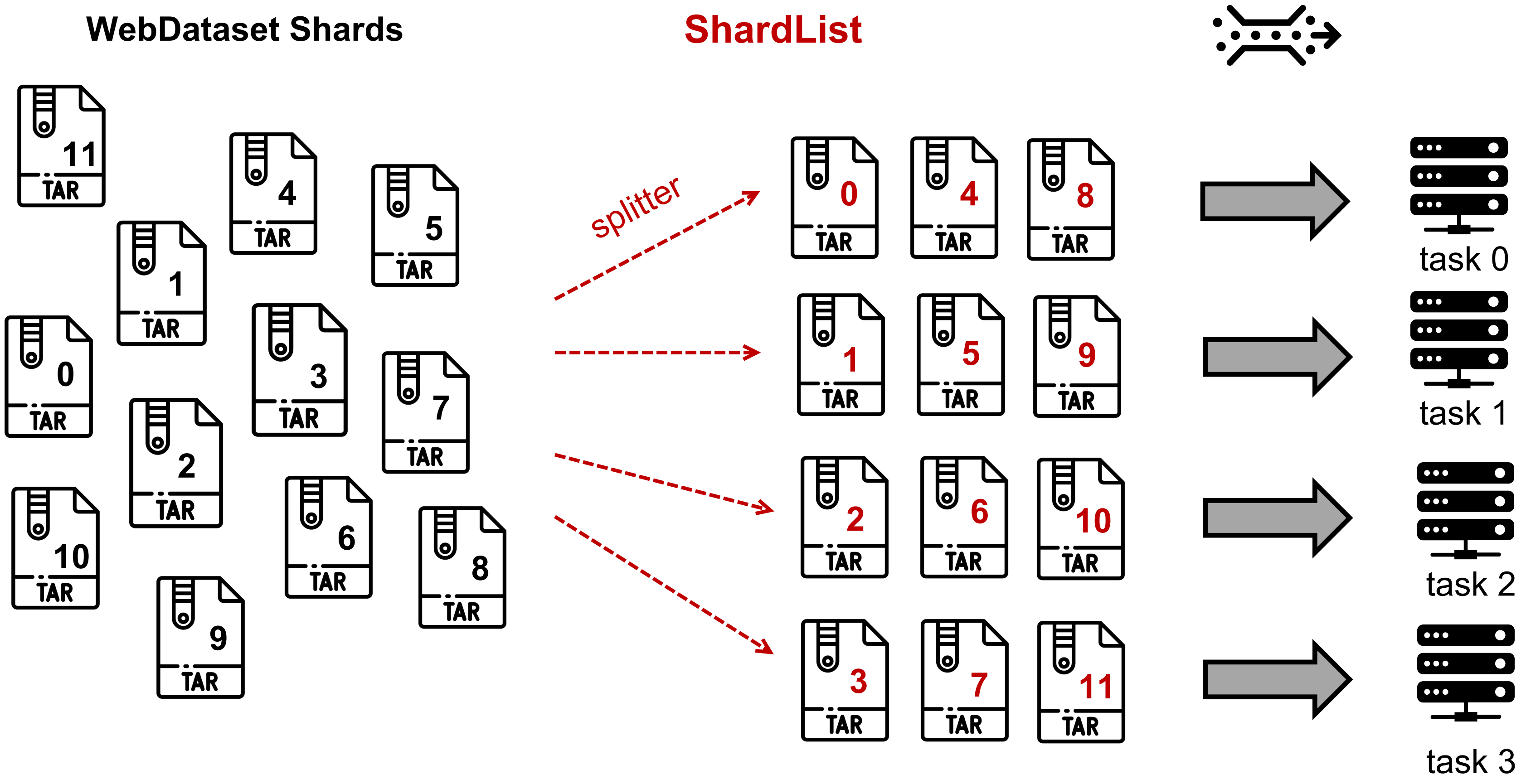
# WebDataset format - More or less shards?



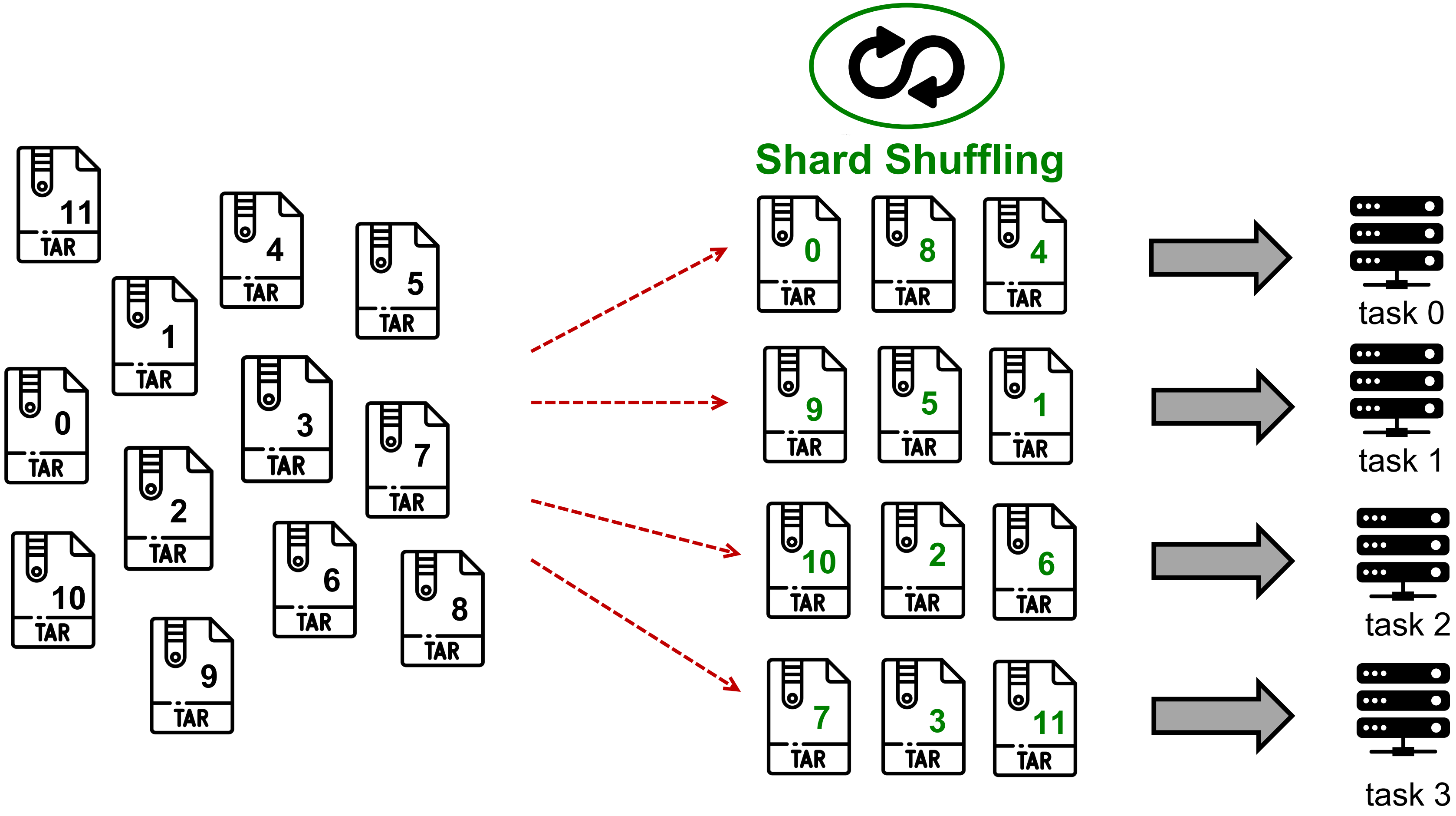
| | More shards | Less shards |
|---|---|---|
| Large scale distribution | + | - |
| Shared bandwidth | + | - |
| Inodes quota | - | + |
| Number of I/O | - | + |



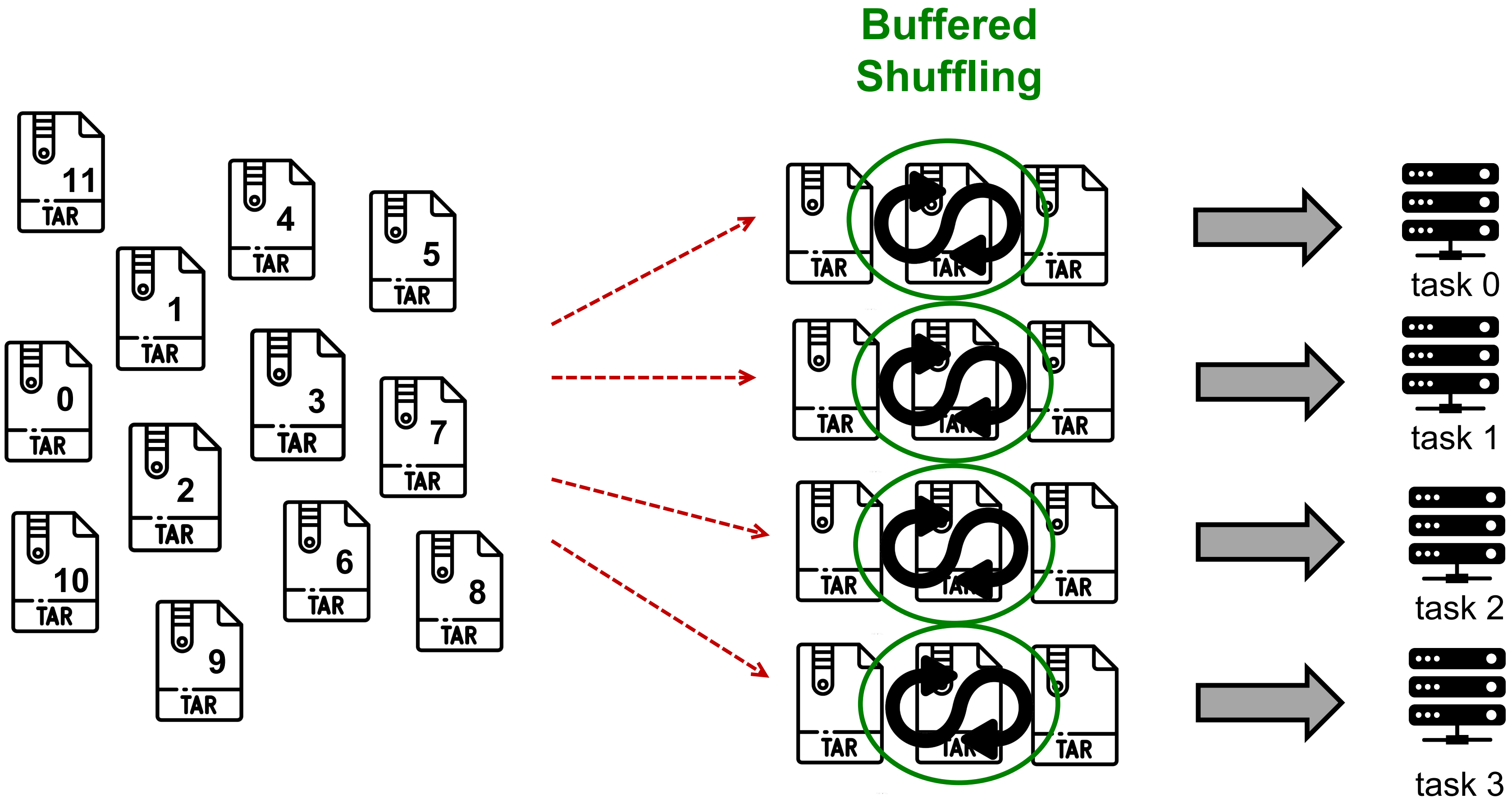
# WebDataset – Multiworker sharding



# WebDataset - Shuffling



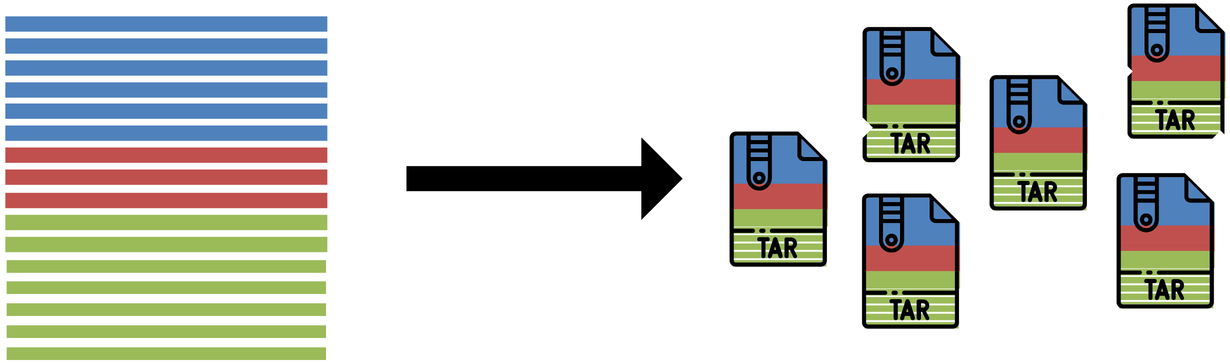
# WebDataset - Shuffling



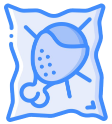
# WebDataset - Generation

When generating WebDataset shards, don't forget to:

- Distribute the samples as **evenly** as possible among the shards.
- Choose the number of shards **according to the number of GPUs** you will use.
- Distribute the samples so that each shard contains a **representative part of the dataset.**




+ Converting data before creating the archives to improve decoding performance?



# WebDataset - Implementation

```
import webdataset as wds

def my_splitter(paths):
    paths = list(paths)
    return paths[idr_torch.rank::idr_torch.world_size]

paths = os.environ['DSDIR']+'/imagenet/webdataset/imagenet_train-{000000..000127}.tar'
train_dataset_len = 1281167
train_dataset = wds.WebDataset(paths, nodesplitter=my_splitter, shardshuffle=True)\
                                .shuffle(1000)\
                                .decode("torchrgb")\
                                .to_tuple('input.pyd', 'output.pyd')\
                                .map_tuple(transform, lambda x: x)\
                                .batched(mini_batch_size)\
                                .with_length(train_dataset_len)

nbatches = train_dataset_len // global_batch_size
train_loader = wds.WebLoader(train_dataset, batch_size=None,\
                             num_workers=num_workers,\
                             persistent_workers=persistent_workers,\
                             pin_memory=pin_memory,\
                             prefetch_factor=prefetch_factor\
                             ).slice(nbatches)
train_loader.length = nbatches
```

# WebDataset - Implementation

```python
import webdataset as wds

def my_splitter(paths):
    paths = list(paths)
    return paths[idr_torch.rank::idr_torch.world_size]
```

distribute shards among processes

```python
paths = os.environ['DSDIR']+'/imagenet/webdataset/imagenet_train-{000000..000127}.tar'
train_dataset_len = 1281167
train_dataset = wds.WebDataset(paths, nodesplitter=my_splitter, shardshuffle=True)\
                                .shuffle(1000)\
                                .decode("torchrgb")\
                                .to_tuple('input.pyd', 'output.pyd')\
                                .map_tuple(transform, lambda x: x)\
                                .batched(mini_batch_size)\
                                .with_length(train_dataset_len)

nbatches = train_dataset_len // global_batch_size
train_loader = wds.WebLoader(train_dataset, batch_size=None,\
                             num_workers=num_workers,\
                             persistent_workers=persistent_workers,\
                             pin_memory=pin_memory,\
                             prefetch_factor=prefetch_factor\
                             ).slice(nbatches)
train_loader.length = nbatches
```

# WebDataset - Implementation

```
import webdataset as wds

def my_splitter(paths):
    paths = list(paths)
    return paths[idr_torch.rank::idr_torch.world_size]

paths = os.environ['DSDIR']+'/imagenet/webdataset/imagenet_train-{000000..000127}.tar'
train_dataset_len = 1281167
train_dataset = wds.WebDataset(paths, nodesplitter=my_splitter, shardshuffle=True)\
                                .shuffle(1000)\
                                .decode("torchrgb")\
                                .to_tuple('input.pyd', 'output.pyd')\
                                .map_tuple(transform, lambda x: x)\
                                .batched(mini_batch_size)\
                                .with_length(train_dataset_len)

nbatches = train_dataset_len // global_batch_size
train_loader = wds.WebLoader(train_dataset, batch_size=None,\
                             num_workers=num_workers,\
                             persistent_workers=persistent_workers,\
                             pin_memory=pin_memory,\
                             prefetch_factor=prefetch_factor\
                             ).slice(nbatches)
train_loader.length = nbatches
```

shuffling shards indexes per process

# WebDataset - Implementation

```
import webdataset as wds

def my_splitter(paths):
    paths = list(paths)
    return paths[idr_torch.rank::idr_torch.world_size]

paths = os.environ['DSDIR']+'/imagenet/webdataset/imagenet_train-{000000..000127}.tar'
train_dataset_len = 1281167
train_dataset = wds.WebDataset(paths, nodesplitter=my_splitter, shardshuffle=True)\
                                    .shuffle(1000)\
                                    .decode("torchrgb")\
                                    .to_tuple('input.pyd', 'output.pyd')\
                                    .map_tuple(transform, lambda x: x)\
                                    .batched(mini_batch_size)\
                                    .with_length(train_dataset_len)

nbatches = train_dataset_len // global_batch_size
train_loader = wds.WebLoader(train_dataset, batch_size=None,\
                             num_workers=num_workers,\
                             persistent_workers=persistent_workers,\
                             pin_memory=pin_memory,\
                             prefetch_factor=prefetch_factor\
                             ).slice(nbatches)
train_loader.length = nbatches
```

shuffling samples per process

# WebDataset - Implementation

```
import webdataset as wds

def my_splitter(paths):
    paths = list(paths)
    return paths[idr_torch.rank::idr_torch.world_size]

paths = os.environ['DSDIR']+'/imagenet/webdataset/imagenet_train-{000000..000127}.tar'
train_dataset_len = 1281167
train_dataset = wds.WebDataset(paths, nodesplitter=my_splitter, shardshuffle=True)\
                                .shuffle(1000)\
                                .decode("torchrgb")\
                                .to_tuple('input.pyd', 'output.pyd')\
                                .map_tuple(transform, lambda x: x)\
                                .batched(mini_batch_size)\
                                .with_length(train_dataset_len)

nbatches = train_dataset_len // global_batch_size
train_loader = wds.WebLoader(train_dataset, batch_size=None,\
                             num_workers=num_workers,\
                             persistent_workers=persistent_workers,\
                             pin_memory=pin_memory,\
                             prefetch_factor=prefetch_factor\
                             ).slice(nbatches)
train_loader.length = nbatches
```

} description of shard content

# WebDataset - Implementation

```python
import webdataset as wds

def my_splitter(paths):
    paths = list(paths)
    return paths[idr_torch.rank::idr_torch.world_size]

paths = os.environ['DSDIR']+'/imagenet/webdataset/imagenet_train-{000000..000127}.tar'
train_dataset_len = 1281167
train_dataset = wds.WebDataset(paths, nodesplitter=my_splitter, shardshuffle=True)\
                                .shuffle(1000)\
                                .decode("torchrgb")\
                                .to_tuple('input.pyd', 'output.pyd')\
                                .map_tuple(transform, lambda x: x)\
                                .batched(mini_batch_size)\
                                .with_length(train_dataset_len)

nbatches = train_dataset_len // global_batch_size
train_loader = wds.WebLoader(train_dataset, batch_size=None,\
                             num_workers=num_workers,\
                             persistent_workers=persistent_workers,\
                             pin_memory=pin_memory,\
                             prefetch_factor=prefetch_factor\
                             ).slice(nbatches)
train_loader.length = nbatches
```

transforming and batching

# WebDataset - Implementation

```
import webdataset as wds

def my_splitter(paths):
    paths = list(paths)
    return paths[idr_torch.rank::idr_torch.world_size]

paths = os.environ['DSDIR']+'/imagenet/webdataset/imagenet_train-{000000..000127}.tar'
train_dataset_len = 1281167
train_dataset = wds.WebDataset(paths, nodesplitter=my_splitter, shardshuffle=True)\
                                .shuffle(1000)\
                                .decode("torchrgb")\
                                .to_tuple('input.pyd', 'output.pyd')\
                                .map_tuple(transform, lambda x: x)\
                                .batched(mini_batch_size)\
                                .with_length(train_dataset_len)

nbatches = train_dataset_len // global_batch_size
train_loader = wds.WebLoader(train_dataset, batch_size=None,\
                             num_workers=num_workers,\
                             persistent_workers=persistent_workers,\
                             pin_memory=pin_memory,\
                             prefetch_factor=prefetch_factor\
                             ).slice(nbatches)
train_loader.length = nbatches
```

define len(train_dataset)

# WebDataset - Implementation

```python
import webdataset as wds

def my_splitter(paths):
    paths = list(paths)
    return paths[idr_torch.rank::idr_torch.world_size]

paths = os.environ['DSDIR']+'/imagenet/webdataset/imagenet_train-{000000..000127}.tar'
train_dataset_len = 1281167
train_dataset = wds.WebDataset(paths, nodesplitter=my_splitter, shardshuffle=True)\
                                .shuffle(1000)\
                                .decode("torchrgb")\
                                .to_tuple('input.pyd', 'output.pyd')\
                                .map_tuple(transform, lambda x: x)\
                                .batched(mini_batch_size)\
                                .with_length(train_dataset_len)

nbatches = train_dataset_len // global_batch_size
train_loader = wds.WebLoader(train_dataset, batch_size=None,\
                             num_workers=num_workers,\
                             persistent_workers=persistent_workers,\
                             pin_memory=pin_memory,\
                             prefetch_factor=prefetch_factor\
                             ).slice(nbatches)
train_loader.length = nbatches
```

batching handled by WebDataset class

# WebDataset - Implementation

```
import webdataset as wds

def my_splitter(paths):
    paths = list(paths)
    return paths[idr_torch.rank::idr_torch.world_size]

paths = os.environ['DSDIR']+'/imagenet/webdataset/imagenet_train-{000000..000127}.tar'
train_dataset_len = 1281167
train_dataset = wds.WebDataset(paths, nodesplitter=my_splitter, shardshuffle=True)\
                                .shuffle(1000)\
                                .decode("torchrgb")\
                                .to_tuple('input.pyd', 'output.pyd')\
                                .map_tuple(transform, lambda x: x)\
                                .batched(mini_batch_size)\
                                .with_length(train_dataset_len)

nbatches = train_dataset_len // global_batch_size
train_loader = wds.WebLoader(train_dataset, batch_size=None,\
                             num_workers=num_workers,\
                             persistent_workers=persistent_workers,\
                             pin_memory=pin_memory,\
                             prefetch_factor=prefetch_factor\
                             ).slice(nbatches)
train_loader.length = nbatches
```

usual DataLoader args

# WebDataset - Implementation

```
import webdataset as wds

def my_splitter(paths):
    paths = list(paths)
    return paths[idr_torch.rank::idr_torch.world_size]

paths = os.environ['DSDIR']+'/imagenet/webdataset/imagenet_train-{000000..000127}.tar'
train_dataset_len = 1281167
train_dataset = wds.WebDataset(paths, nodesplitter=my_splitter, shardshuffle=True)\
                                .shuffle(1000)\
                                .decode("torchrgb")\
                                .to_tuple('input.pyd', 'output.pyd')\
                                .map_tuple(transform, lambda x: x)\
                                .batched(mini_batch_size)\
                                .with_length(train_dataset_len)

nbatches = train_dataset_len // global_batch_size
train_loader = wds.WebLoader(train_dataset, batch_size=None,\
                             num_workers=num_workers,\
                             persistent_workers=persistent_workers,\
                             pin_memory=pin_memory,\
                             prefetch_factor=prefetch_factor\
                             ).slice(nbatches)
train_loader.length = nbatches
```

drop_last equivalent

# WebDataset - Implementation

```
import webdataset as wds

def my_splitter(paths):
    paths = list(paths)
    return paths[idr_torch.rank::idr_torch.world_size]

paths = os.environ['DSDIR']+'/imagenet/webdataset/imagenet_train-{000000..000127}.tar'
train_dataset_len = 1281167
train_dataset = wds.WebDataset(paths, nodesplitter=my_splitter, shardshuffle=True)\
                                .shuffle(1000)\
                                .decode("torchrgb")\
                                .to_tuple('input.pyd', 'output.pyd')\
                                .map_tuple(transform, lambda x: x)\
                                .batched(mini_batch_size)\
                                .with_length(train_dataset_len)

nbatches = train_dataset_len // global_batch_size
train_loader = wds.WebLoader(train_dataset, batch_size=None,\
                             num_workers=num_workers,\
                             persistent_workers=persistent_workers,\
                             pin_memory=pin_memory,\
                             prefetch_factor=prefetch_factor\
                             ).slice(nbatches)
train_loader.length = nbatches
```

define len(train_loader)

# WebDataset - Perfomance test

I/O loop over the dataset
(calculation-free iterations)

```
start_time = datetime.datetime.now()

for i, (images,labels) in enumerate(loader):
    print(f'{i} / {nb_batches}', end="\r")

end_time = datetime.datetime.now()
delta_time = (end_time_it - start_time_it).total_seconds()
```

- Execution on 1 GPU

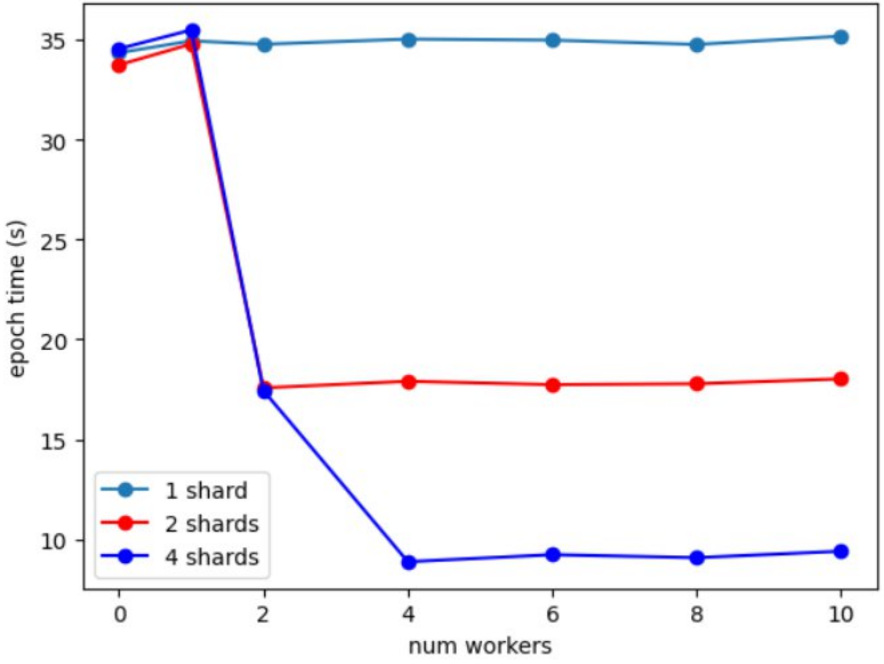
# WebDataset - Perfomance test

I/O loop over the dataset
(calculation-free iterations)

CIFAR10 ~ 50k images

- Sharding is necessary to benefit from parallel implementation (DataLoader multi-processing).

CIFAR 10 (50k images)
elapsed time - 1 epoch



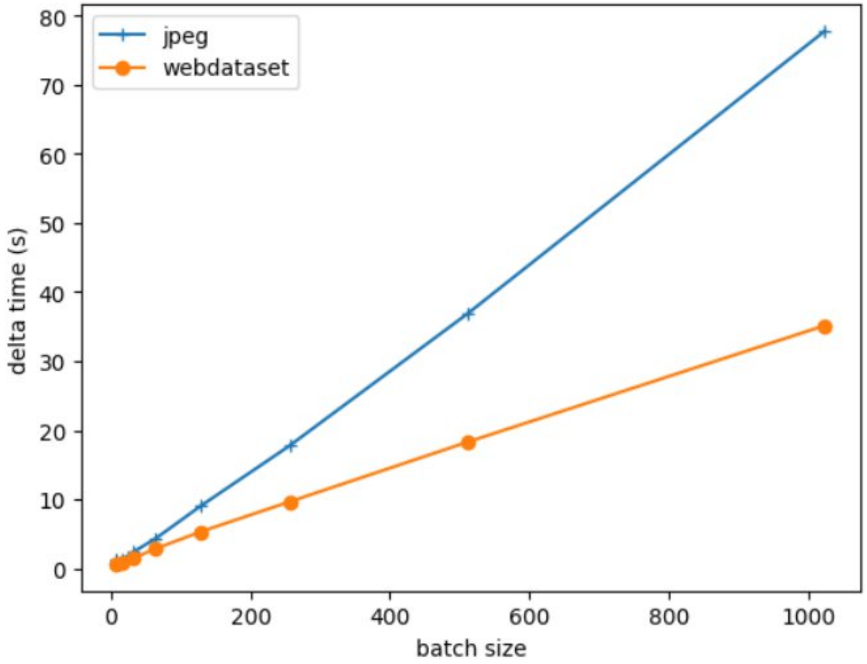
# WebDataset - Perfomance test

I/O loop over the dataset
(calculation-free iterations)

Imagenet ~ 1.3M images
128 shards ~10k images per shard (+labels)
1 shard (images + labels) ~ 6GB

- The more samples are needed per batch, the more efficient is the WebDataset format (fewer I/Os).

Imagenet
elapsed time - 10 iterations




more I/O generated

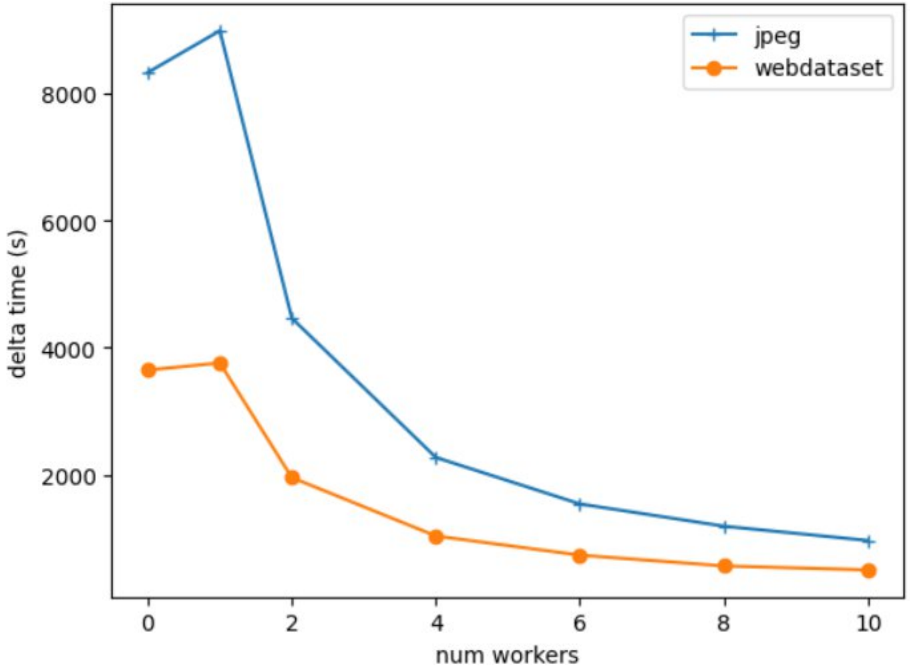
# WebDataset - Perfomance test

I/O loop over the dataset
(calculation-free iterations)

Imagenet ~ 1.3M images
128 shards ~10k images per shard (+labels)
1 shard (images + labels) ~ 2GB

- The WebDataset format scales up.

Imagenet
elapsed time - 1 epoch



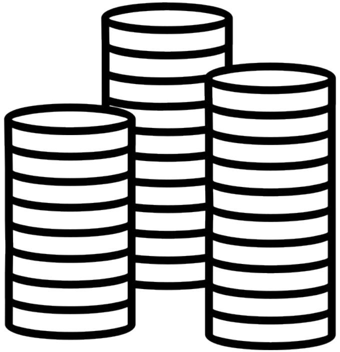
# WebDataset - Perfomance test

A complete training over the Imagenet dataset (dlojz.py)

| | Original jpeg dataset | WebDataset format |
|---|---|---|
| Elapsed time (41 epochs) | 30min43s | 29min56s |
| Test accuracy | 72% | 72% |

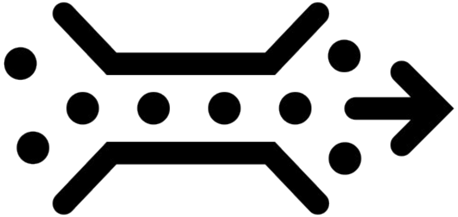
# Conclusion

**Storage Disks**



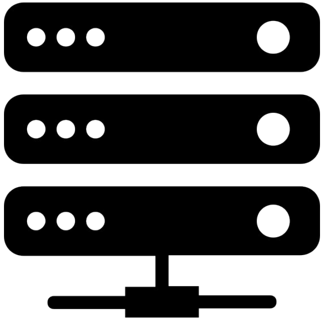
1. I/O performance

**Interconnection Network**
Omnipath



2. Shared Bandwidth

**CPU workers**

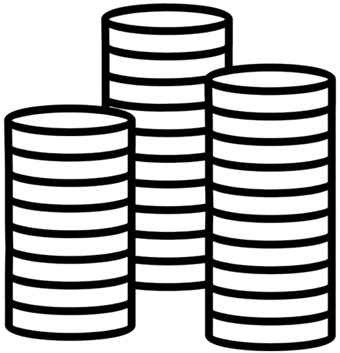


3. Decoder performance

- Disk spaces: WORK / DSDIR or SCRATCH
- Data format: binary or compressed
- Dataset format: alternative format like WebDataset

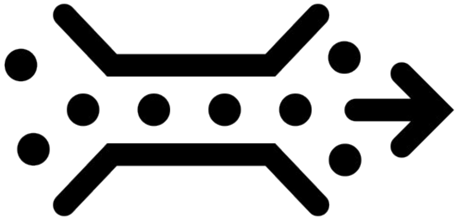
# Conclusion

**Storage Disks**



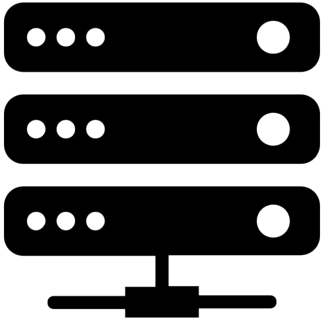
1. I/O performance

**Interconnection Network**
Omnipath



2. Shared Bandwidth

**CPU workers**

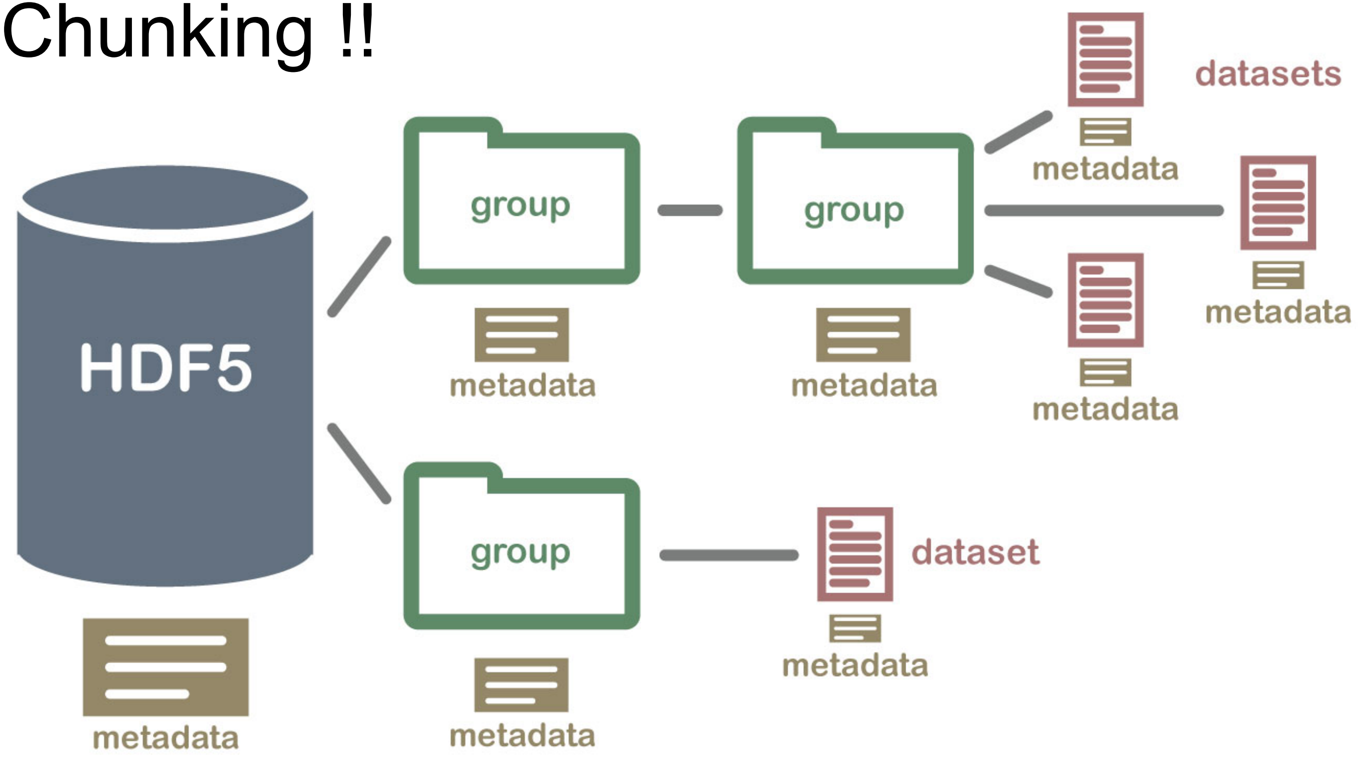


3. Decoder performance

- Disk spaces: WORK / DSDIR or SCRATCH
- Data format: binary or compressed
- Dataset format: alternative format like WebDataset

# Other Formats – HDF5

with Chunking !!



# Other Formats – Parquet

with Chunking !!

Better for 2-dimensional table

| | Column 1 | Column 2 | Column 3 | Column 4 | Column 5 |
|---|---|---|---|---|---|
| | Product | Customer | Country | Date | Sales Amount |
| Row Group 1 | Ball | John Doe | USA | 2023-01-01 | 100 |
| | T-Shirt | John Doe | USA | 2023-01-02 | 200 |
| Row Group 2 | Socks | Maria Adams | UK | 2023-01-01 | 300 |
| | Socks | Antonio Grant | USA | 2023-01-03 | 100 |
| Row Group 3 | T-Shirt | Maria Adams | UK | 2023-01-02 | 500 |
| | Socks | John Doe | USA | 2023-01-05 | 200 |

# HuggingFace Datasets

Hugging Face Hub

`dataset = load_dataset("dataset_name")`, get any of these datasets ready to use in a dataloader for training/evaluating a ML model (Numpy/Pandas/PyTorch/TensorFlow/JAX) - **from remote access or from local copy**.

Two types of dataset objects: **Dataset** or **IterableDataset** .

- **IterableDataset** is ideal for big datasets (think hundreds of GBs!)
- **Dataset** is great for everything else.

General :

- **In-memory data (dictionary, Pandas DataFrames, generator)**
- CSV
- JSON
- **Parquet**
- Arrow
- SQL
- **WebDataset**

Audio :

- Local Files Dictionary
- AudioFolder
- AudioFolder with metadata

Text :

- Text Files list
- TextFolder

Vision :

- Local Files Dictionary
- ImageFolder
- **WebDataset**

Tabular :

- CSV files
- Pandas DataFrames
- Databases (SQLite, PostgreSQL)

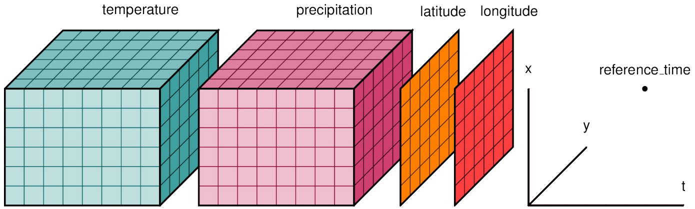
https://huggingface.co/docs/datasets/index

# ESPRI-IA Use Case


## Context : Large Training Scientific Dataset

**NetCDF** (network Common Data Form) is a file format for **storing multidimensional scientific data** (variables) such as temperature, humidity, pressure, wind speed, and direction.

**Xarray** is a library for working with domain-agnostic data-structures, labeled arrays, NetCDF, Zarr, ...




## Test :

**Storage format :** Numpy, HDF5, WebDataset, Zarr

Zarr is a high-level storage format
Dataset-level abstraction with indexing

**High-performance Compressor :** BLOSC + LZ4

BLOSC is a meta-Compressor

Slide: https://espri.ipsl.fr/wp-content/uploads/2023/12/storage_formats.pdf
video: https://www.youtube.com/watch?v=w8TJcBf87zw

# ESPRI-IA Use Case


## Conclusion:

**with BLOSC + LZ4, loading (I/O + com + decoding) compressed data is faster than loading uncompressed data! Recommended for WebDataset and Zarr!**

For Short Dataset:
Numpy/Pickle is the best suitable storage format !!

For Long Dataset:

- Map style: **Zarr >** HDF5
- Iterable: **WebDataset >** Zarr ≈ HDF5

Slide: https://espri.ipsl.fr/wp-content/uploads/2023/12/storage_formats.pdf
video: https://www.youtube.com/watch?v=w8TJcBf87zw

# Attempt at Standardization

- TorchData?

PyTorch

Get Started | Ecosystem | Mobile | Blog | Tutorials | Docs | Resources | GitHub

0.6.0 ▾

Search Docs

API Reference:

Iterable-style DataPipes

Map-style DataPipes

Utility Functions

## TorchData (see note below on current status)

What is TorchData? | Stateful DataLoader | Install guide | Contributing | License

⚠️ **June 2024 Status Update: Removing DataPipes and DataLoader V2**

**We are re-focusing the torchdata repo to be an iterative enhancement of torch.utils.data.DataLoader. We do not plan on continuing development or maintaining the [ `DataPipes` ] and [ `DataLoaderV2` ] solutions, and they will be removed from the torchdata repo. We'll also be revisiting the `DataPipes` references in pytorch/pytorch. In release 0.8.0 (July 2024) they will be marked as deprecated, and in 0.9.0 (Oct 2024) they will be deleted. Existing users are advised to pin to `torchdata==0.8.0` or an older version until they are able to migrate away. Subsequent releases will not include DataPipes or DataLoaderV2. The old version of this README is available here. Please reach out if you suggestions or comments (please use #1196 for feedback).**

- MLCommons/Croissant

## Summary

Croissant 🥐 is a high-level format for machine learning datasets that combines metadata, resource file descriptions, data structure, and default ML semantics into a single file; it works with existing datasets to make them easier to find, use, and support with tools. Croissant builds on schema.org, and its Dataset vocabulary, a widely used format to represent datasets on the Web, and make them searchable. You can find a gentle introduction in the companion paper Croissant: A Metadata Format for ML-Ready Datasets.