ESPRI-IA

STORAGE FORMATS

Sébastien Gardoll
May - 2023

ESPRI-IA

Mission

Facilitating advanced AI adoption within IPSL laboratories

Assistance in the form of:

- Engineering, methodological and technical consulting
- Relay of training events
- Technology watch

Regarding:

- Machine Learning in general and Deep Learning in particular
- Data engineering
- Computing facilities
- Software engineering
- Parallel and distributed computing

Contact: https://tinyurl.com/slack-espri-ia
Website: https://espri.ipsl.fr/services/artificial-intelligence-support/
Abonnement newsletter webinaire : https://tinyurl.com/newsletter-espri-ia

ESPRI-IA

Storage formats

Use case

Use case 1/2

Requirements

Very large training dataset in Deep Learning (DL):

- Dataset can be too big to be loaded in the RAM/VRAM of a node.
- Costly online data extraction make us to create training dataset.
- Numeric, float32 or float64 multidimensional matrices.
- Training dataset is read multiple time during training (epoch).
- Looking for a storage format that:
  - Provides random read of chunks (e.g. images) of the training dataset.
  - Otherwise sequentially read randomly selected partitions of pre-randomized chunks (but less entropy).

Common:

- Dataset size quota & data ingress ⇒ eventually compress data.
- File number quota ⇒ eventually concatenate files (e.g. tar file).
- Minimization of read/write time (IO overhead).

ESPRI-IA

Use cases 2/2

Abstraction

The chunks by chunks case, read/write chunks of dataset:

- Sequentially
- Randomly ⇔ Most frequent case in Deep Learning

⇒ File based chunking: 1 chunk = 1 file. But files may be concatenated into one file with tar or zip, seamlessly!

Example: in computer vision Deep Learning, a chunk is an image and each image is a file.

The partitions by partitions case, read/write partitions of chunks of dataset, then iterate chunks:

- Sequentially
- Randomly

⇒ 1 Partition = 1 file ⇒ storage format provides chunk access within the partition.

e.g. In climate modelling, a chunk can be a grid of values of a physical variable at a specified time step. The grids are grouped according to days, months or years. Example ERA5, each file contains one month of gridded values of a variable.

ESPRI-IA

Storage formats in Python 1/2

Simplified overview

Multidimensional and self descriptive low level storage formats for numerical data in Python:

- Numpy
- HDF5 studied
- Zarr
- Parquet (does imply JVM serialization ?)
- Tiff
- Etc.

Frameworks supporting geospatial features (date, geolocalization, variable level, etc.):

- NetCDF4: NetCDF4 ⇒ HDF5
- Xarray: NetCDF4 over HDF5
- Xarray: NetCDF over Zarr compatibility ?
- NCZarr: NetCDF4.8 over Zarr
- Geotiff
- Etc.

Databases supporting geospatial features:

- postGIS over postgreSQL
- Etc.

ESPRI-IA

Storage formats

HDF5, Numpy and Zarr

Data selection

Abstraction levels are not equivalent

HDF5 and Numpy are at file level. The selection data ⇔ selection of files is left up to you (based on directory path and file name).

Selection of data:

- Selection of files in FS (compute paths and file names)
- Open the files
- Selection of data within these fileS (slicing and eventually concatenation)
- Close the files

Zarr is at dataset level, data can be stored in directories, zip file, MongoDB, Redis, N5, HadoopFS, Amazon S3, etc. Data storage is abstracted. For example, no more struggle when the selection is straddling two files!

Selection of data:

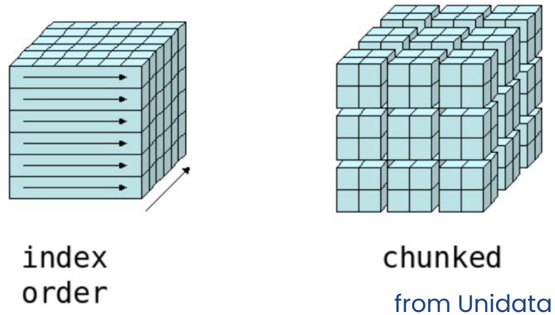
- Open the dataset (load just the metadata)
- Slicing an array like Numpy (orthogonal slicing is supported): load in RAM only the chunks containing the selected data.

Data store:

- Zarr-dir: Zarr with directory as data storage.
- Zarr-zip: Zarr with a zip file as data storage.

Chunking

Different approaches



from Unidata

Concepts:

- File based chunking: 1 chunk = 1 file. The whole file is loaded into RAM.
- Chunking within file or N chunks grouped into 1 file. Only the part of the file that corresponds to the chunk is loaded into RAM.

Numpy:

- File based chunking is straight forward. Chunk/file selection is left up to you.
- Chunking within file is possible with mmap but low level (byte offset management left up to you).

HDF5:

- File based chunking is straight forward. Chunk/file selection is left up to you.
- Chunking within file: offsetted sequential read of chunks is supported but random read.

Zarr:

- Is designed on file based chunking.
- Provided indexed array like Numpy ⇒ Random read made easy

Chunk shape is critical for IO performances: they must fit to all applications!

Storage formats in Python 2/2

Features

Container	Compression	Concatenation	Chunking	Concurrent chunk write	Embedded metadata	Cloud stores
Numpy	✓ with third parties	✓ with Tar but loose random read	✓ complicated with mmap	✓ different chunks/files ✗ same chunk/file	✗	✗
HDF5	✓ std & third parties	✓ with Tar but loose random read	✓ sequential read ✗ random read	✓ different chunks/files ✗ same chunk/file	✓	✗
Zarr	✓ with third parties	✓ with zipstore	✓ file based	✓ different chunks/files ✗ same chunk/file	✓	✓

ESPRI-IA

Storage formats

Experiments

Method 1/2

Dataset

- Dataset size: 8268.86 MBytes. Sequential online extraction: 59 s vs loading: 15.8 s
- Chunks:
  - 896, around 9.22 MBytes/chunk.
  - Chunk: stack of 3 matrices 560x1440 of float 32 bits, from ERA5:
    - Mean Sea Level pressure: 101020.42 ± 1224.54 Pa
    - Total Column Water Vapor: 22.98 ± 5.66 kg/m^2
    - U component wind speed at 10 meter above the sea: 0.14 ± 5.66 m/s
- This dataset is big enough to be a partition of a dataset ⇒ The same dataset can be used for chunks by chunks or partitions by partitions experiments.
- Dataset is build using Numpy ndarray: most frequent cases ⇒ Keep in mind that Numpy, as a storage format, is favored in this study!

Method 2/2

Experimental protocol

- The experiments are performed on the same hardware (spiritx48-2).
- Read/write on local file system: prevent network overhead and concurrency (/tmp).
- Read/write on SSD: most frequent case in Deep Learning (HAL and Jean Zay) ; HDD has lower access time and getting worst when random read (i.e. blocks, sectors and platters).
- Blocks of files in Linux cache are systematically evicted before and after reads or writes (command vmtouch).
- Every experiment is performed 3 times, only means are considered.
- Experiment is redone if its RSD > 5% (5% rsd ⇔ std = 50 ms for 1 second elapsed time experiment).
- All the chunks of the dataset is read, sequentially or randomly.

Compression codecs

Only lossless ones

- Lossless compression codecs are general-purpose compression algorithm, expert configuration is not necessary as opposed to lossy codecs (fixed scale offset, quantize, bitround, latent space, etc.) which may have better performances. Jpeg and png are lossy image formats and are not general-purpose and/or deal with integers.
- Studied codecs: lzma (xz), gzip, bz2, zlib, zstd, lz4, lz4hc.
- Blosc version: blosc-bloscLZ (FastLZ), blosc-lz4, blosc-lz4hc, blosc-zlib, blosc-zstd.
- Extra codecs for HDF5: sz, sz3, lzf, blosc2 (blosclz, lz4, lz4hc, zlib, zstd).
- Codecs for HDF5 are from package hdf5plugins (Fortran compliance?).
- Codecs for Numpy and Zarr are from package numcodecs (Fortran compliance?).

Blosc

Meta-compressor

- Blosc is not a compression codec:
    - Wrapps an compression codec and speeds up transmission of data to the CPU.
    - Leverages SIMD (SSE2) and multi-threading capabilities present in recent multi-core processors.
    - Chunks data (again!) to fit the size of the CPU caches so as to fight CPU starvation (waiting too long for data).
- Blosc is production ready!
- Blosc is available for C & Python (wrapper). Any Fortran library? Fortran through C library ?
- More informations at https://www.blosc.org/

ESPRI-IA

Storage formats

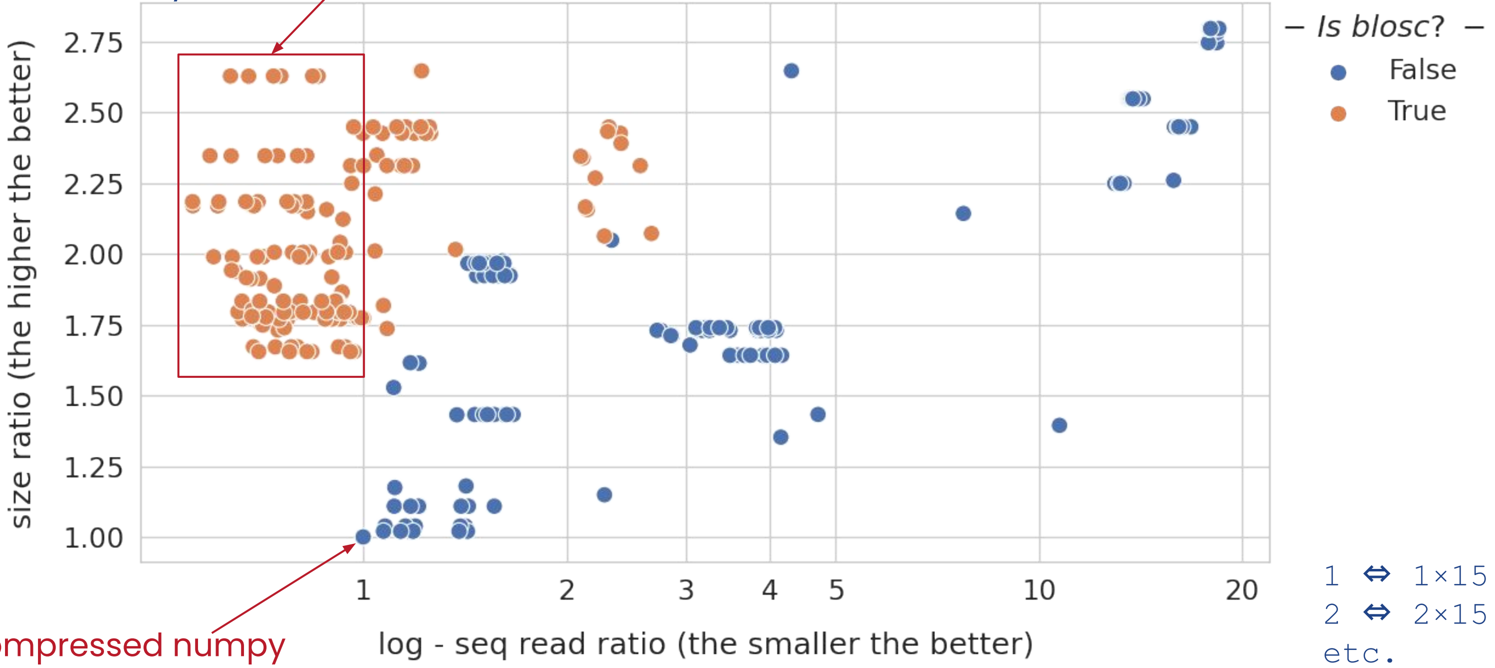
Results

Blosc superiority 1/3

Sequential read

```
1    ⇔ 8268.86÷1    MBytes
1.25 ⇔ 8268.86÷1.25 MBytes
etc.
```

Blosc codecs that have sequential read faster than uncompressed numpy!



Uncompressed numpy

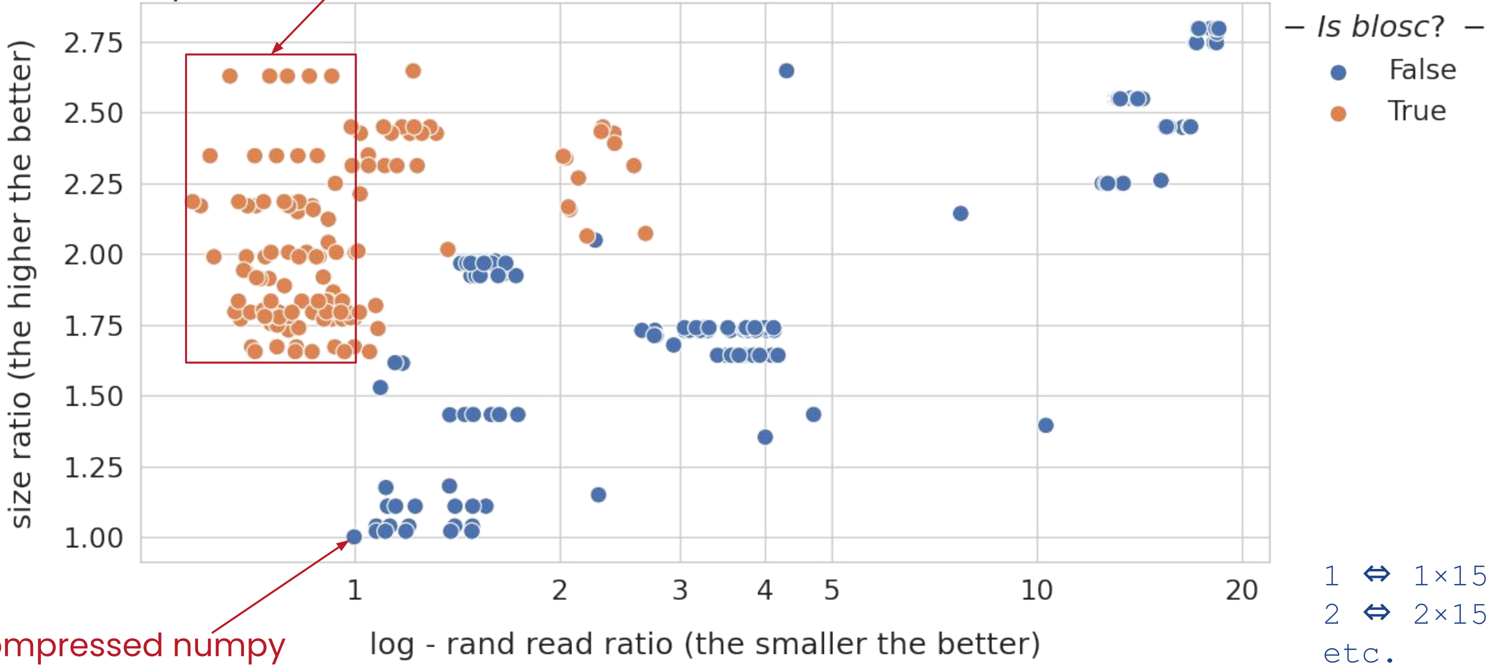
```
1 ⇔ 1×15.8 sec
2 ⇔ 2×15.8 sec
etc.
```

Blosc superiority 2/3

Random read

1 ⇔ 8268.86÷1 MBytes
1.25 ⇔ 8268.86÷1.25 MBytes
etc.

Blosc codecs that have random read faster than uncompressed numpy!



Uncompressed numpy

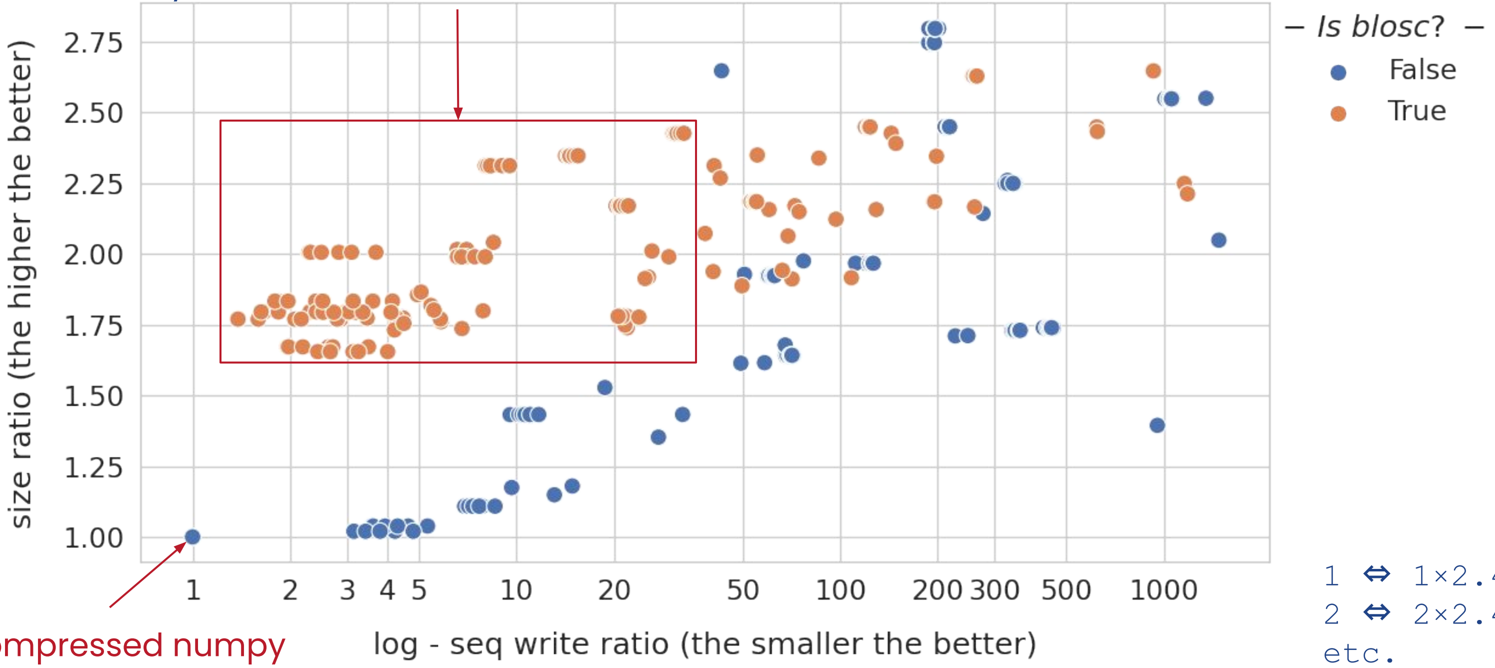
1 ⇔ 1×15.8 sec
2 ⇔ 2×15.8 sec
etc.

ESPRI-IA

Blosc superiority 3/3

Sequential write

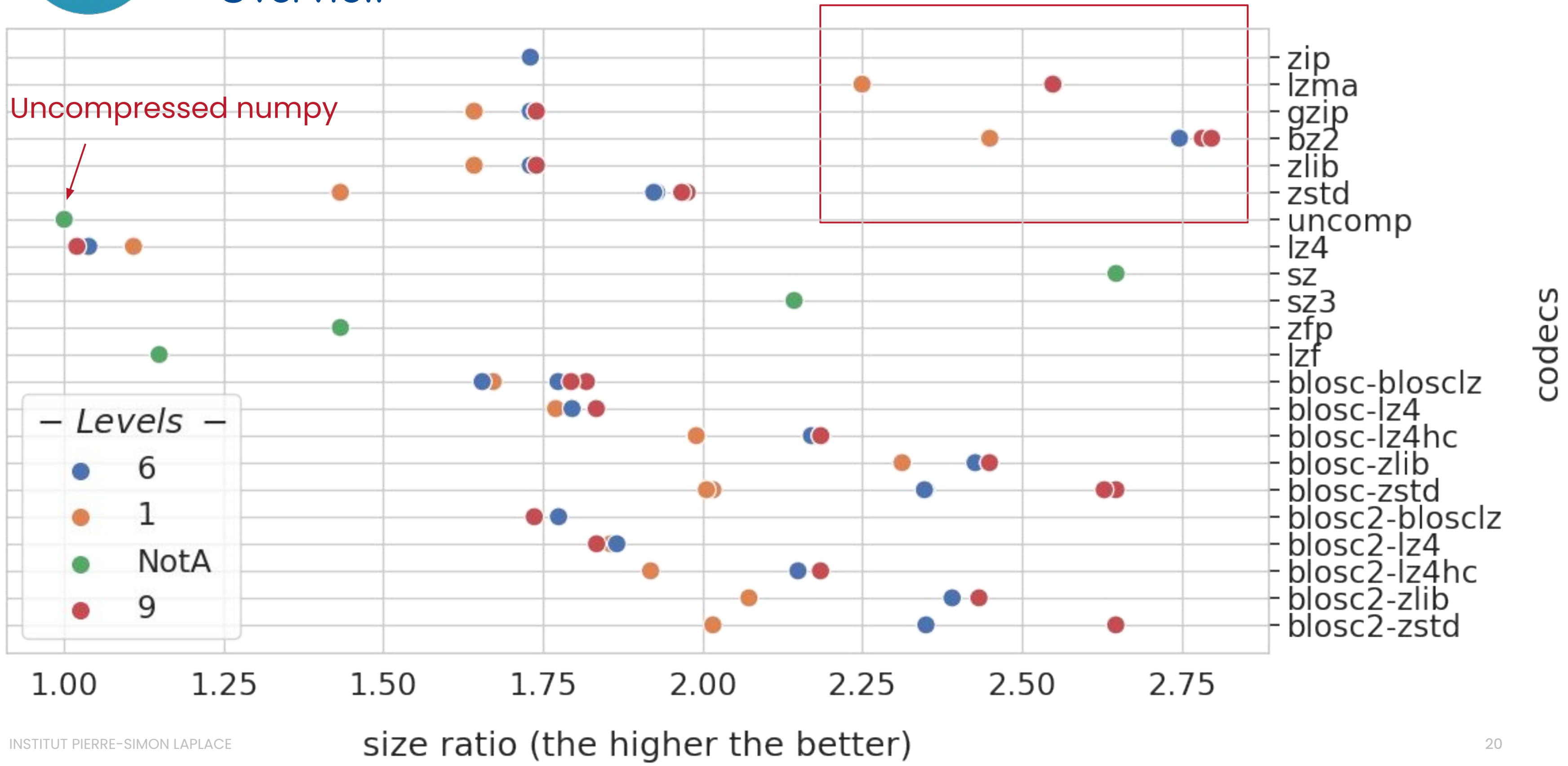
```
1    ⇔ 8268.86÷1    MBytes
1.25 ⇔ 8268.86÷1.25 MBytes
etc.
```



```
1 ⇔ 1×2.45 sec
2 ⇔ 2×2.45 sec
etc.
```

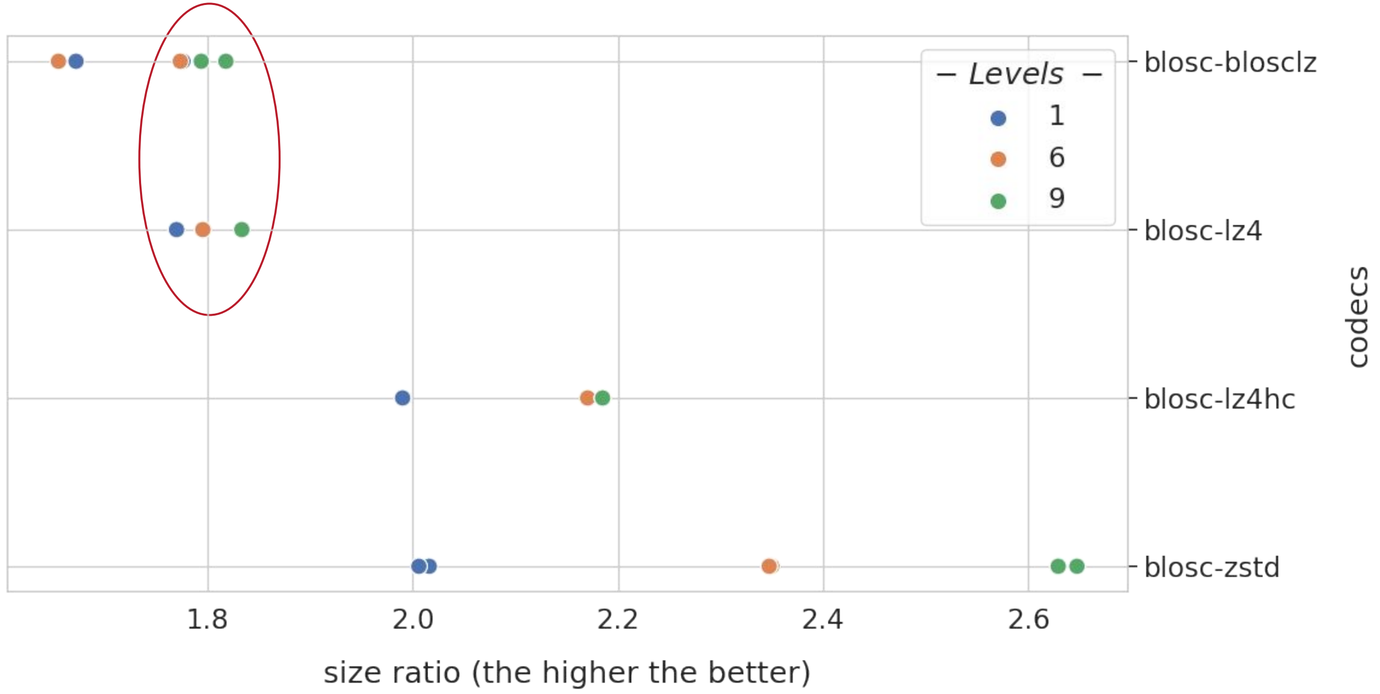
Size ratios 1/2

Overview



Size ratios 2/2

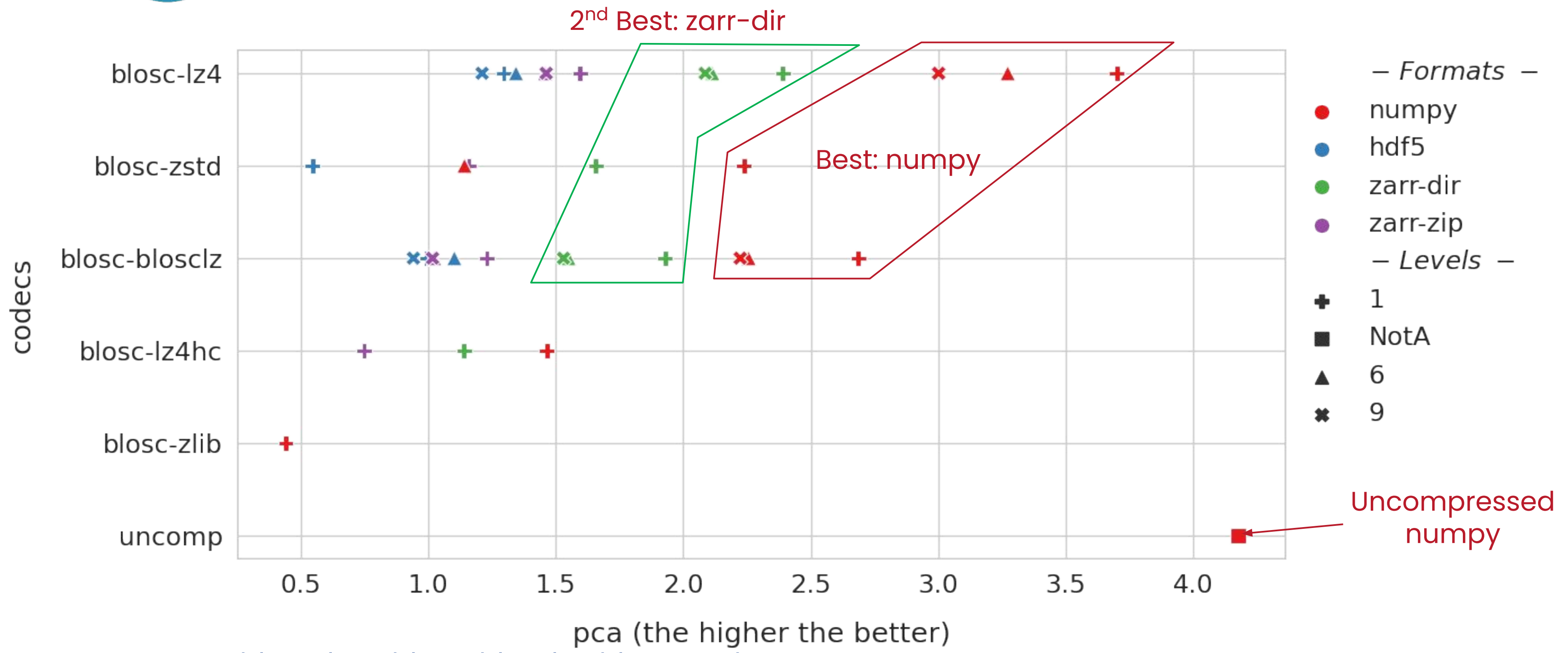
Selected codecs



- blosc-zstd > blosc-lz4hc > blosc-bloscLZ ≈ blosc-lz4
- lz4 has acceleration level not compression

Chunks by chunks 1/4

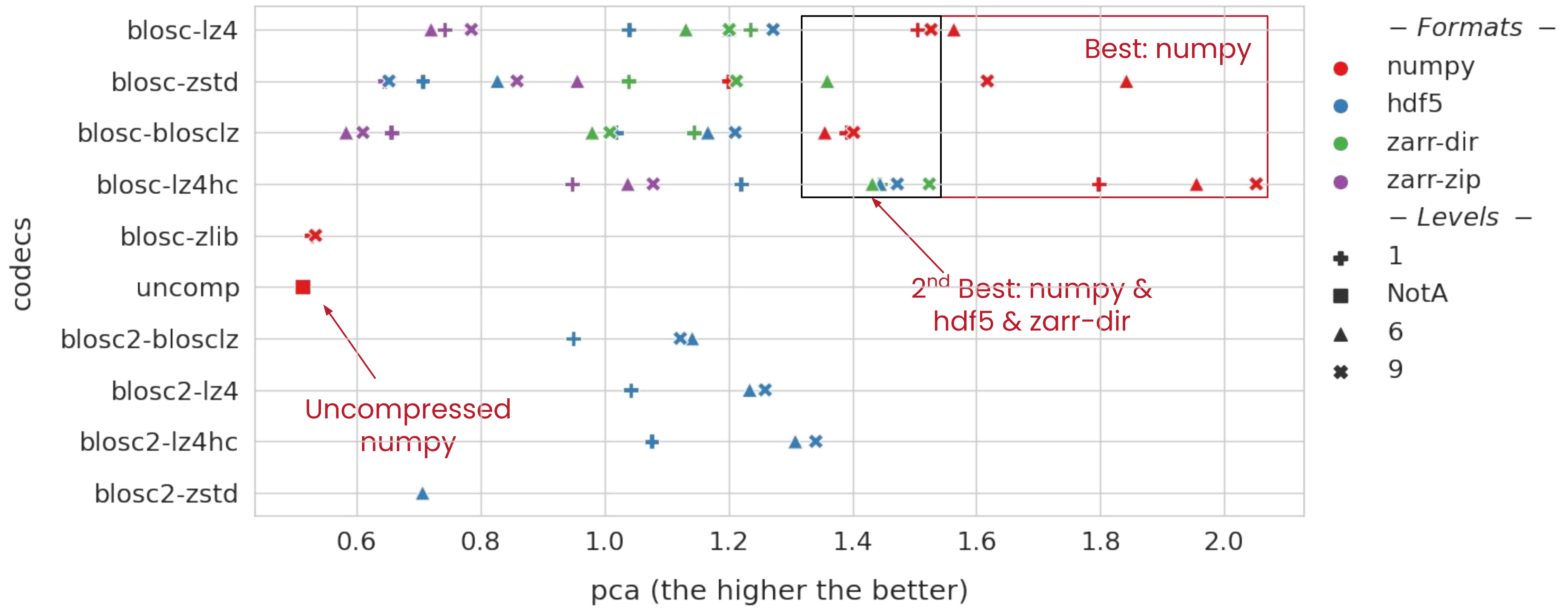
Random read



- blosc-lz4 > blosc-blosclz > blosc-zstd
- numpy > zarr-dir > zarr-zip ≈ hdf5

Chunks by chunks 2/4

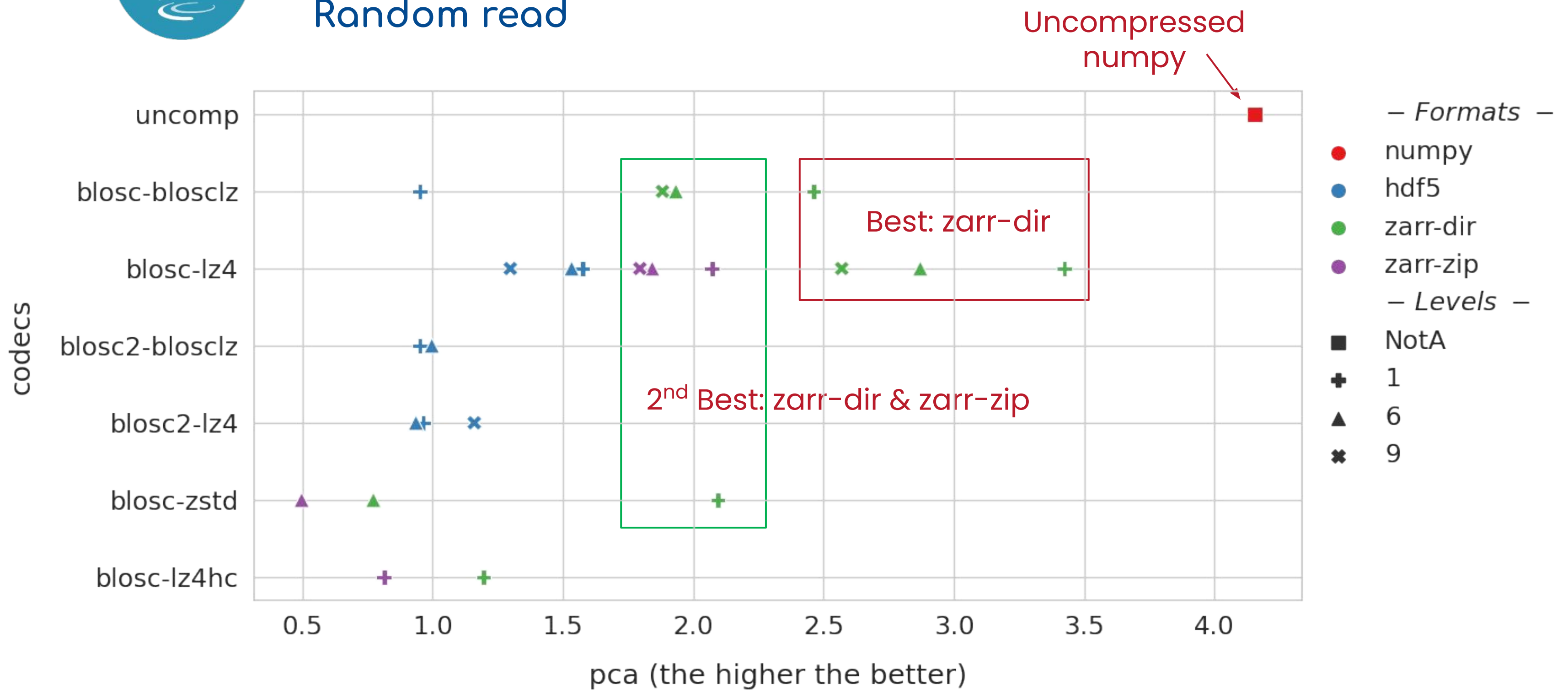
Random read without write performance



- blosc-lz4hc > blosc-zstd >< blosc-lz4
- numpy > hdf5 ≈ zarr-dir > zarr-zip

Partitions by partitions 1/2

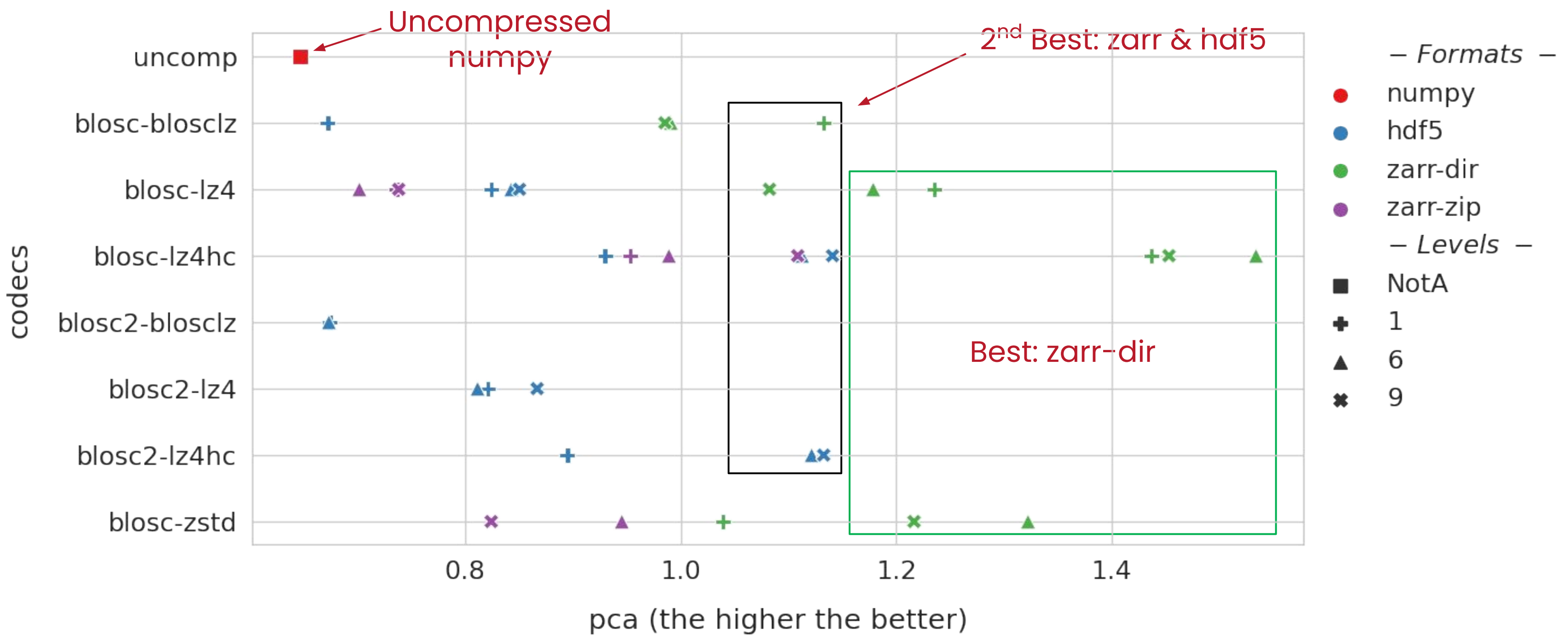
Random read



- lz4 & blosc-* impossible (buffer < 2 GBytes)
- blosc-lz4 > blosc-blosclz > blosc-zstd
- zarr-dir > zarr-zip

Partitions by partitions 2/2

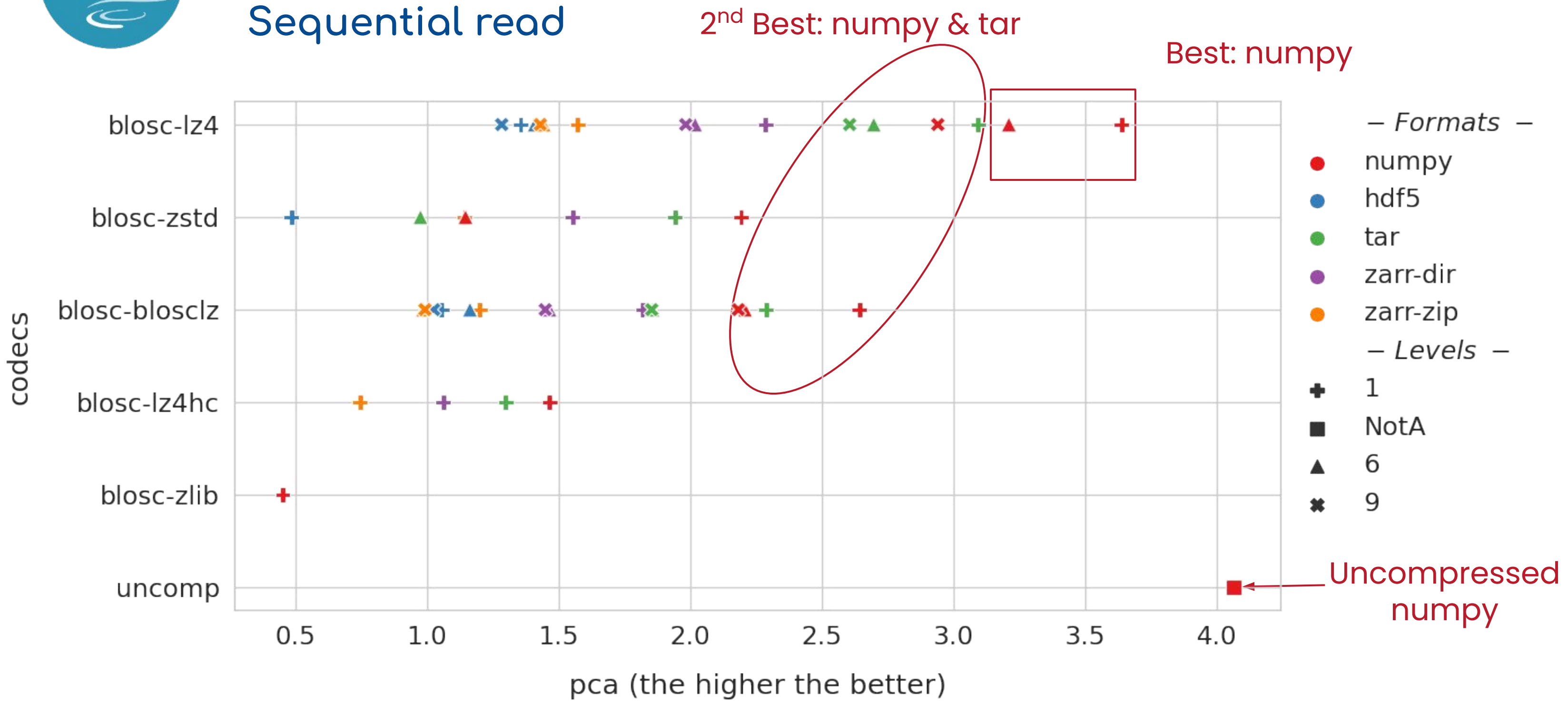
Random read without write performance



- lz4 & blosc-* impossible with numpy (buffer < 2 GBytes)
- blosc-lz4hc > blosc-zstd > blosc-lz4
- zarr-dir > zarr-zip ≈ hdf5

Chunks by chunks 3/4

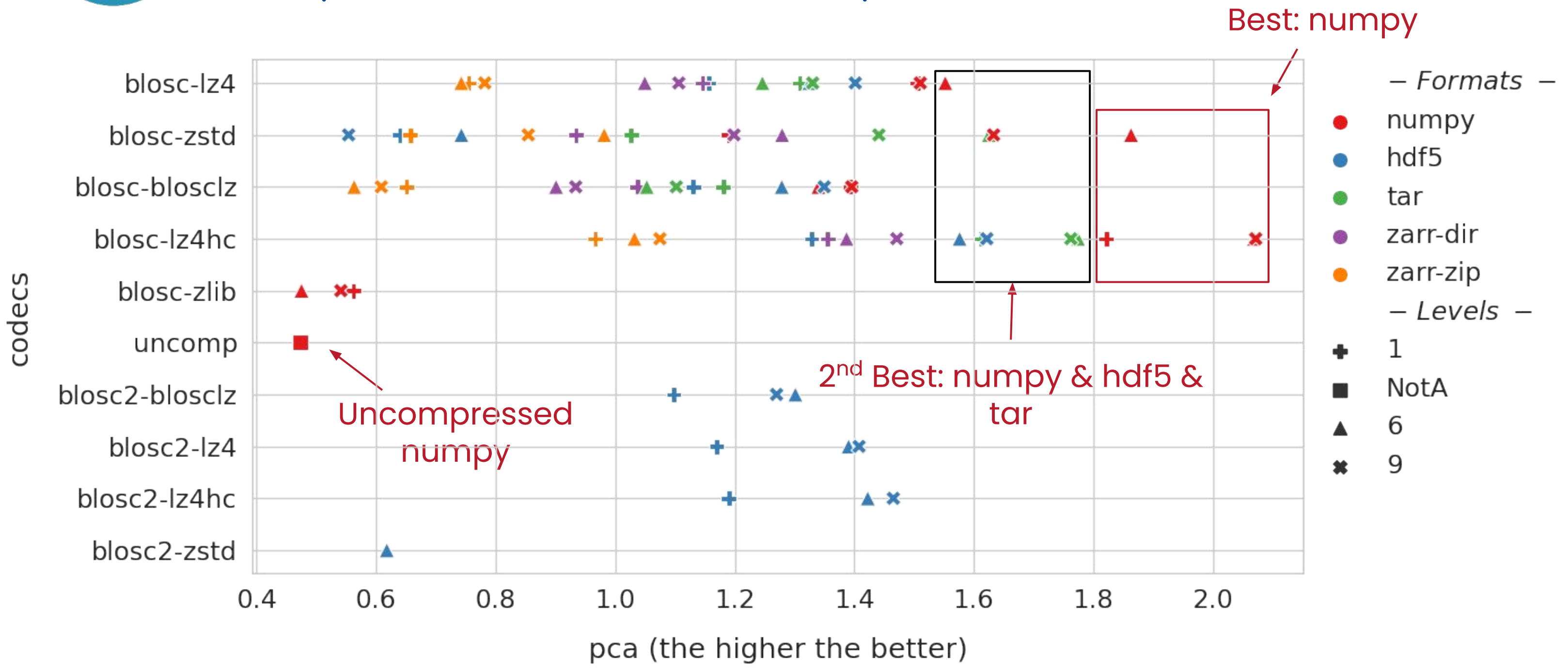
Sequential read



- blosc-lz4 > blosc-blosclz > blosc-zstd
- numpy > tar > zarr-dir

Chunks by chunks 4/4

Sequential read without write performance



- blosc-lz4hc > blosc-zstd > blosc-lz4
- numpy > tar > hdf5

Conclusions

For large training dataset in Deep Learning

- Compression codecs
  - Decompression elapsed time is not related to compression elapsed time.
  - LZMA and bz2 are not suitable for scientific computations.
  - Alway wrap your compression codec with blosc: loading compressed data is faster than loading uncompressed data!
  - Blosc-lz4 seems to be best compromise: default codec for Zarr for good reasons.
  - Blosc-lz4hc is the best option if write overhead doesn't matter so much.
  - Blosc-lz4 and Blosc-blosclz has roughly the same compression ratio.
  - lz4 & blosc are not suitable for large partitions (buffer < 2 GBytes).
- Storage formats
  - Numpy is the best suitable format but you must save the shape and dtype information!
  - If the dataset abstraction is required, Zarr is a good option.
  - If over file quota:
    - Random read: Zarr-zipstore > HDF5 + shuffling & partitioning
    - Sequential read: Numpy + Tar + shuffling & partitioning > Zarr-zipstore ≈ HDF5 + shuffling & partitioning

Perspectives

Suggestions:

- Use or implement chunks loading ahead of time (e.g. workers for Pytorch).
- If write overhead is critical, compress offline.
- Make your own application oriented benchmarks!

Experiment notebooks, results in csv and many more plots available at:

https://gitlab.in2p3.fr/ipsl/espri/espri-ia/projects/storage_formats

Compression tutorial notebook at:

https://ipsl.pages.in2p3.fr/formations/jupyter-notebooks-examples/engineering/storage_formats_for_floats.html

ESPRI-IA

Storage formats

Q & A