Anemoi datasets and data handling

Natalie Zelenka, Florian Pinault, Cathal O’Brien

01

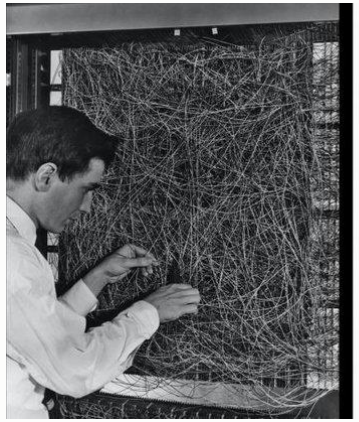
Why data movement matters

Compute has outpaced memory and bandwidth

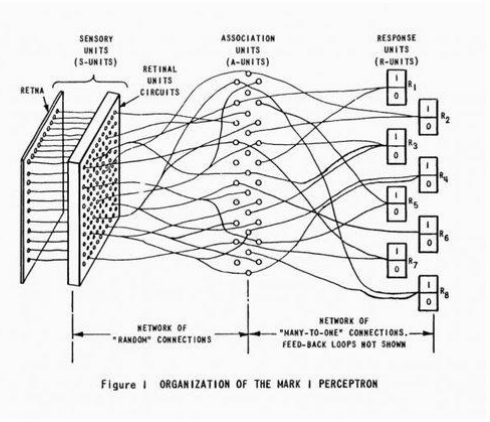
ECMWF

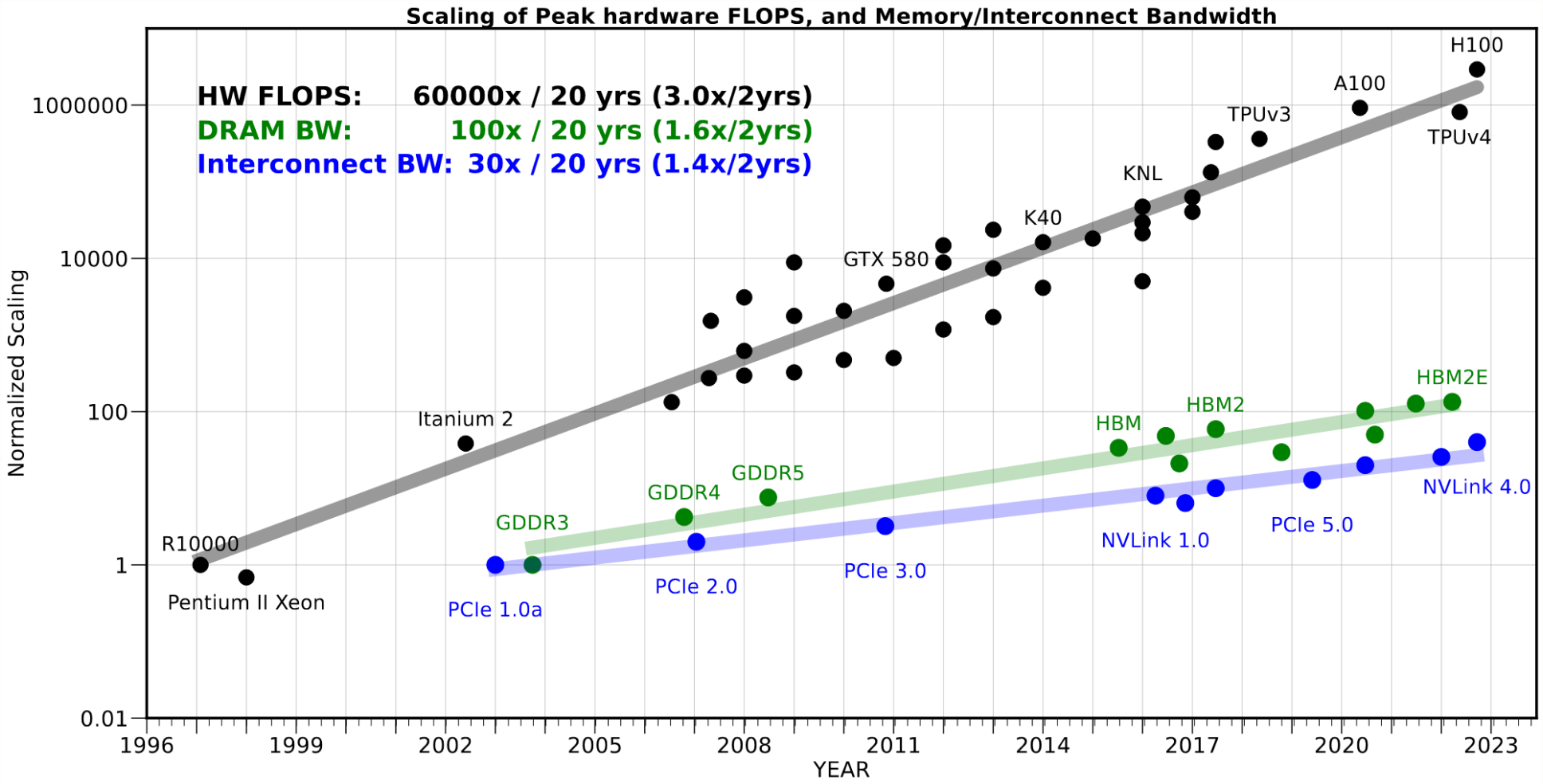
Background

- Machine learning is an old idea
- Increased compute power makes machine learning possible today
  - Moore's law · "Domain-specific compute architectures", e.g. GPUs
- But there is a problem… as compute got faster, other components couldn't keep up



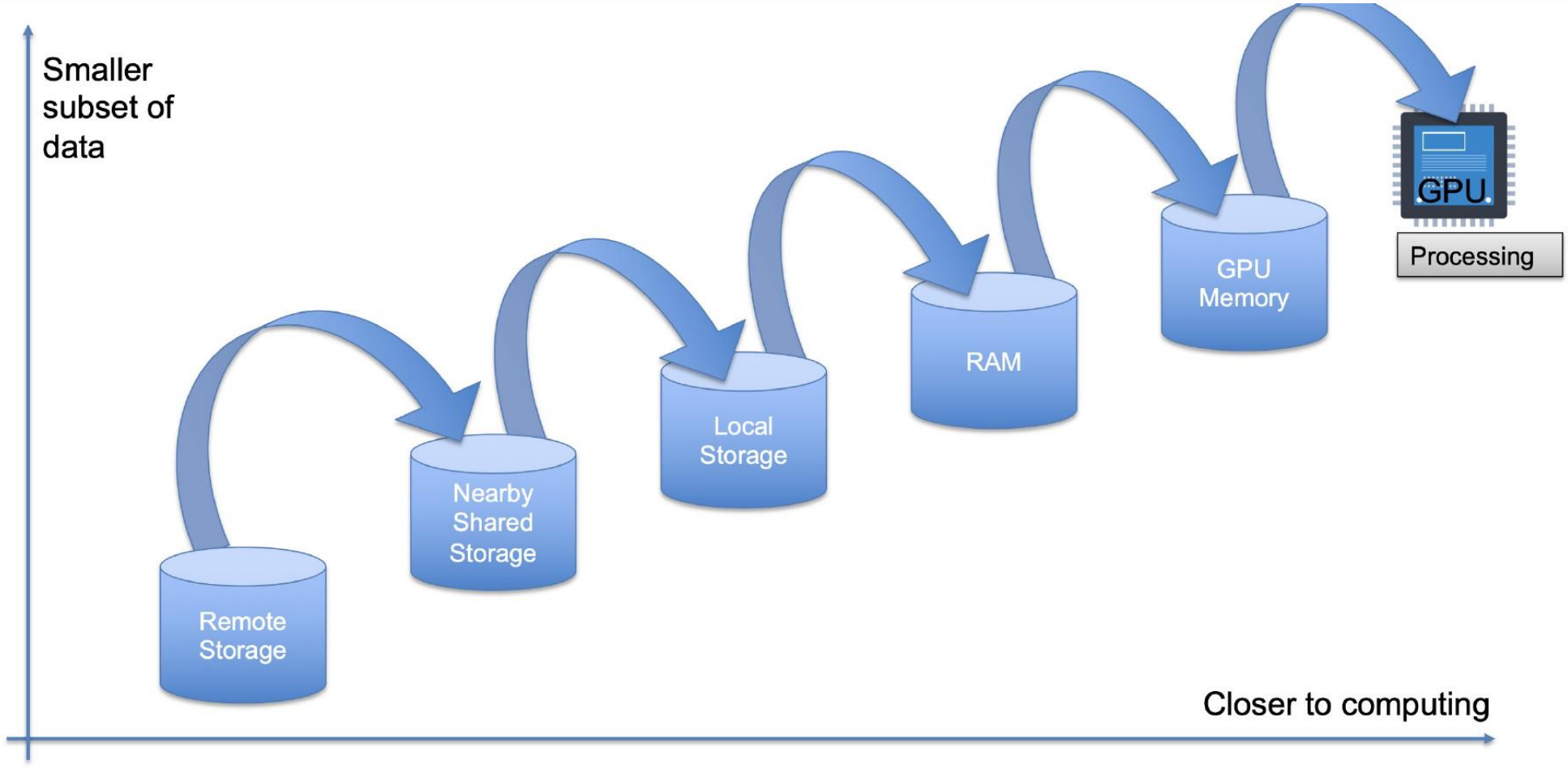
Frank Rosenblatt programming his Perceptron, 1970s



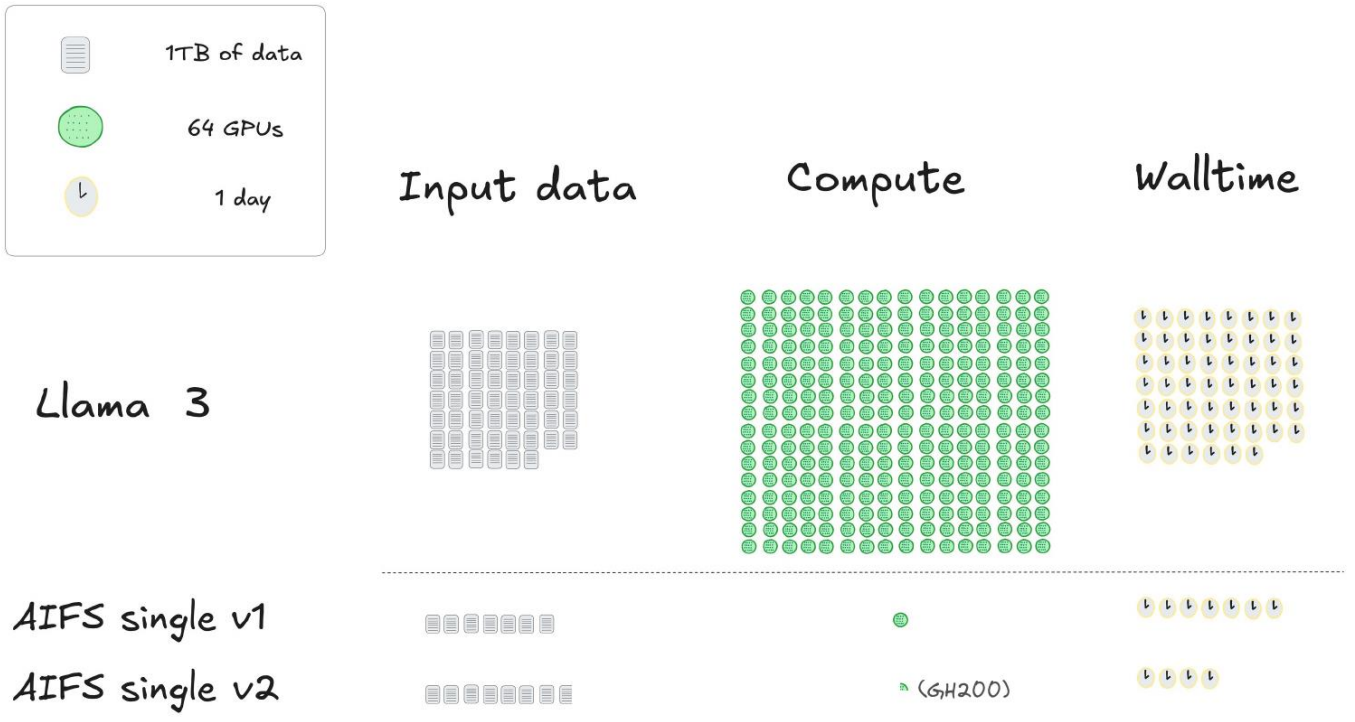


siliconmatter.substack.com/p/the-memory-wall-and-its-implications

Data cascade of caching



Training cost: Llama 3 vs AIFS Single



AIFS Single v1 [1]

Data: **about 7 TB**: ERA5 1979–2022, then IFS analyses 2016–2022
Compute: **64 A100** (40 GB)
Walltime: **about 1 week** ≈ 10,700 GPU-hours

AIFS Single v2 [2]

Data: **about 7.8 TB** (1.12 × v1): ERA5 1979–2022, then IFS analyses 2018–2024, plus 12 new parameters (waves, snow)
Compute: **16 GH200** (120 GB)
Walltime: **about 4 days** ≈ 1,500 GPU-hours

AIFS Single v3

Not published yet

[1] huggingface.co/ecmwf/aifs-single-1.0 (model card, “Training Procedure” and “Hardware”)
[2] huggingface.co/ecmwf/aifs-single-2.0 (model card, “Training Procedure” and “Hardware”)
GPU-hours = GPUs × walltime, computed from the cards. Data sizes are ECMWF estimates, not on the model cards. A100 and GH200 hours are not equivalent.

02

Zarr

How anemoi datasets are stored, and how to use them

ECMWF

Anemoi datasets are stored in Zarr

One Zarr store per dataset, with a few conventions on top

Inside the store

```
era5-o96-1979-2023-6h-v8.zarr/
  data        [dates, variables, ensembles, grid]
  dates       [dates]
  latitudes   [grid]
  longitudes  [grid]
  mean  stdev  minimum  maximum     [variables]
  count  sums  squares  has_nans    [variables]
  .zattrs     variables, frequency, recipe,
              licence, provenance, …
# data: float32, (65744, 101, 1, 40320)
```

Conventions

Name: purpose-content-source-resolution-start-end-frequency-version. Lower case, letters, digits and dashes only.

Chunks by date. A small date (under 25 MB) is one chunk; a bigger one is split in 4 along the grid.

Blosc lz4 compression, level 5. **Zarr format 2** by default; format 3 is experimental.

Statistics are computed while building, and stored with the data.

03

Data layout

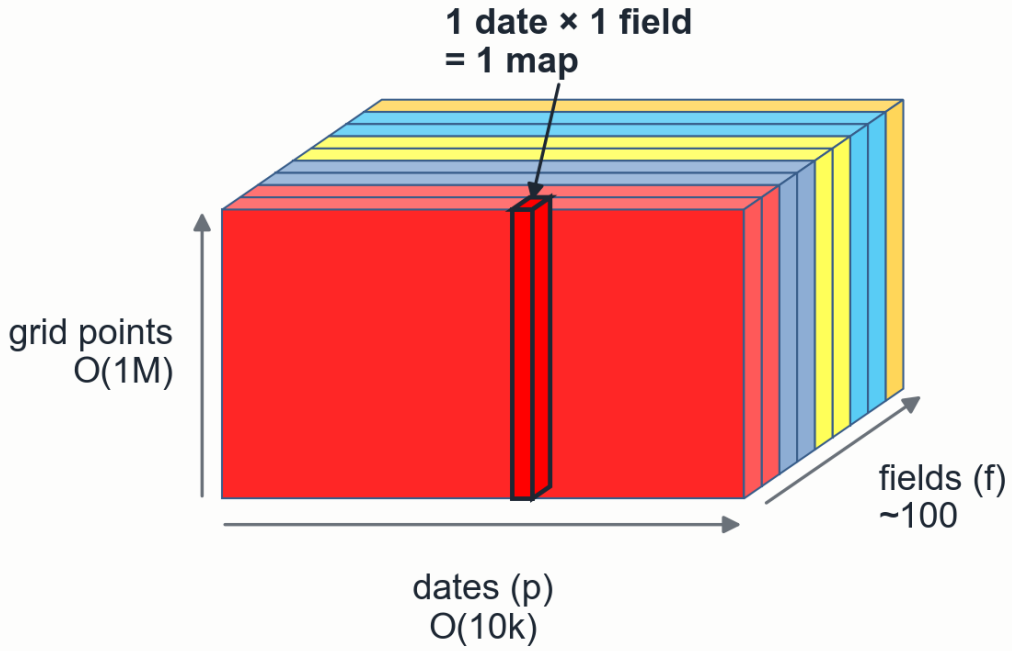
Store data the way it is read

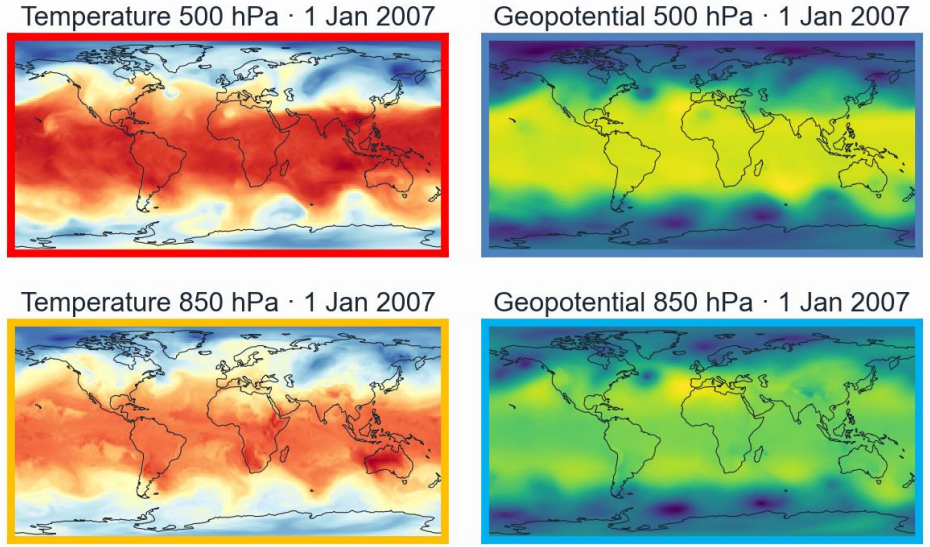
ECMWF

The data is a cube

Three dimensions: dates, fields and grid points (ensemble dimension is ignored here).

One field at one date is a whole map.

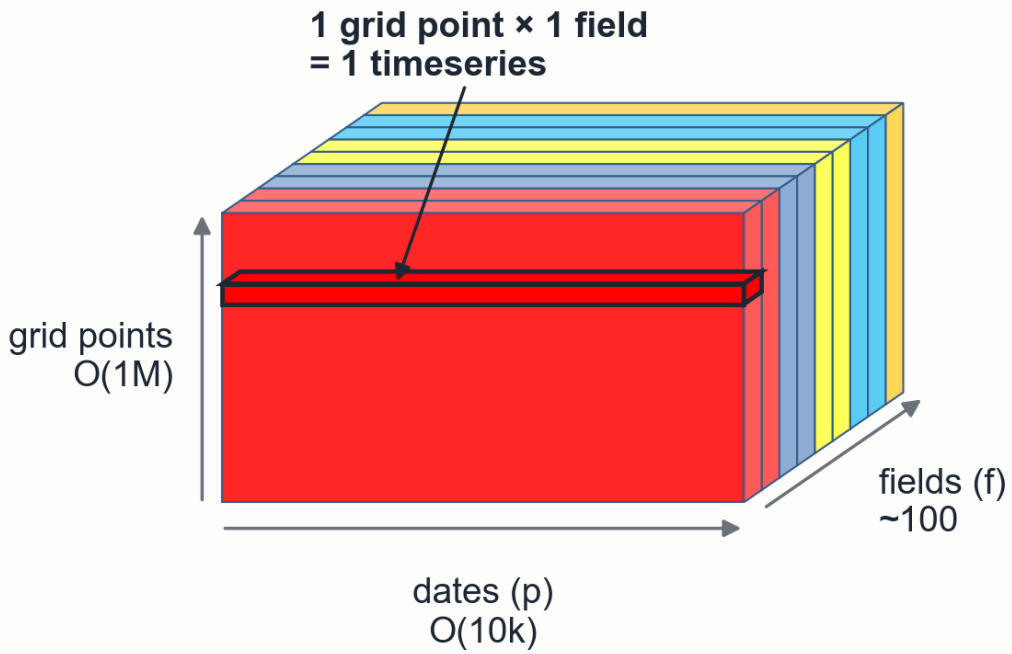


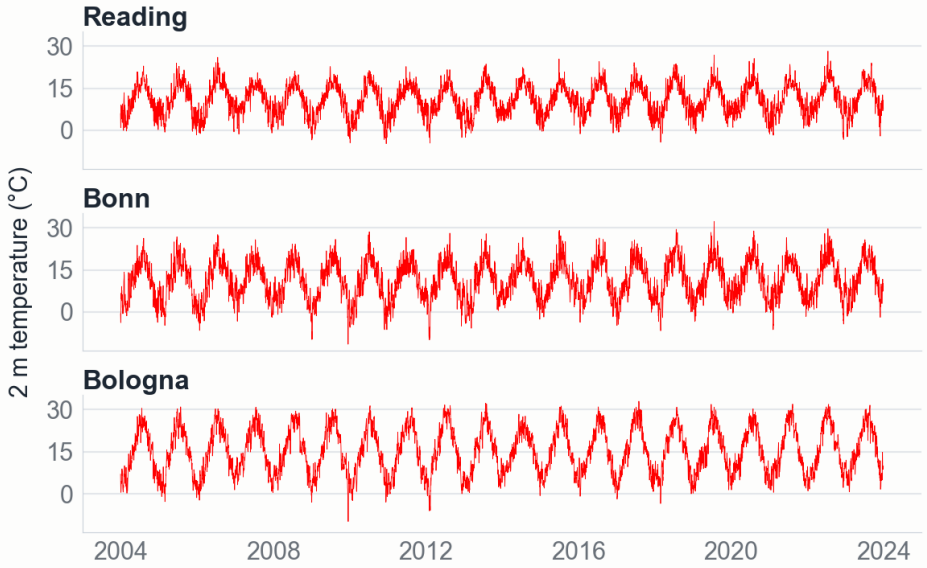


Four of the fields on 1 January 2007. A real dataset has about **100 of them for every date**, each with around **a million grid points**.

The same cube, cut the other way

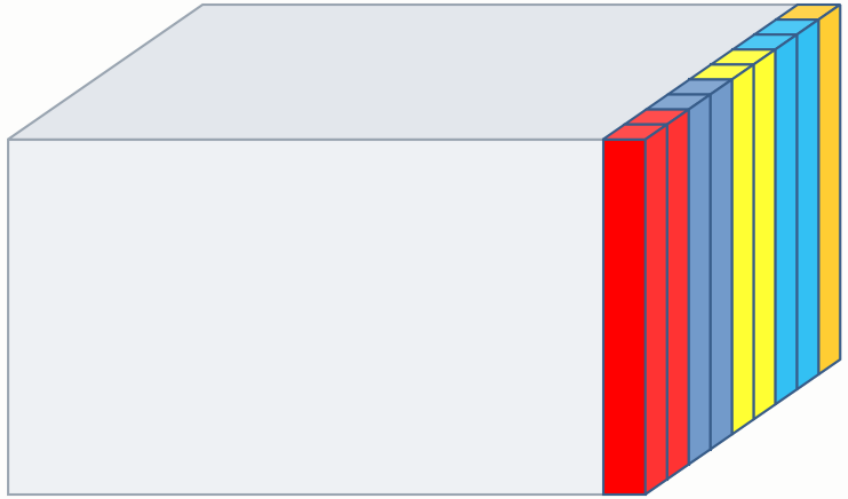
Follow one grid point through all the dates and you get a timeseries

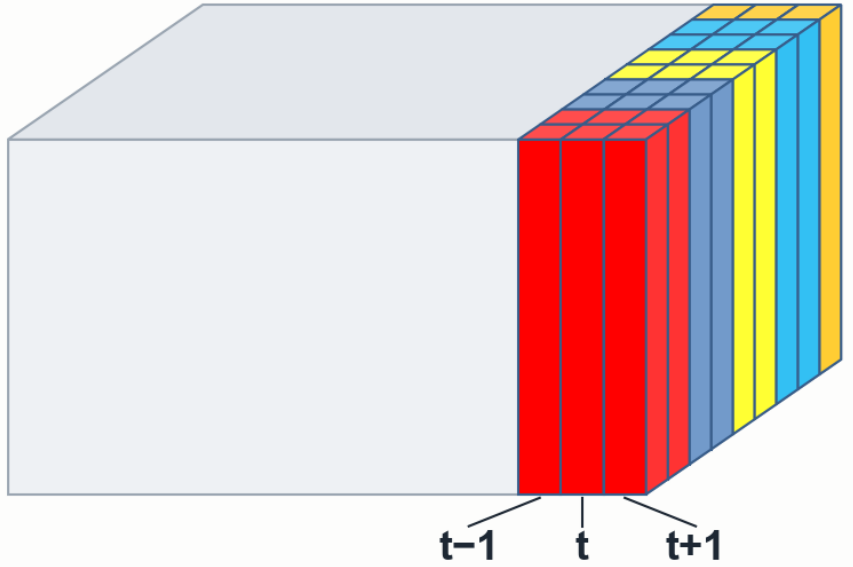




The temperature in Reading, Bonn and Bologna, day by day over 20 years. Each line is **one grid point** of **one field**, read across **all the dates**.

Training a global model: what one sample reads





Read one date

All fields, all grid points

In memory: N320 **≈ 200 MB** · o1280 **≈ 3 GB**
On disk, compressed ~2×: N320 **≈ 100 MB** · o1280 **≈ 1.5 GB**

Read t−1, t, t+1

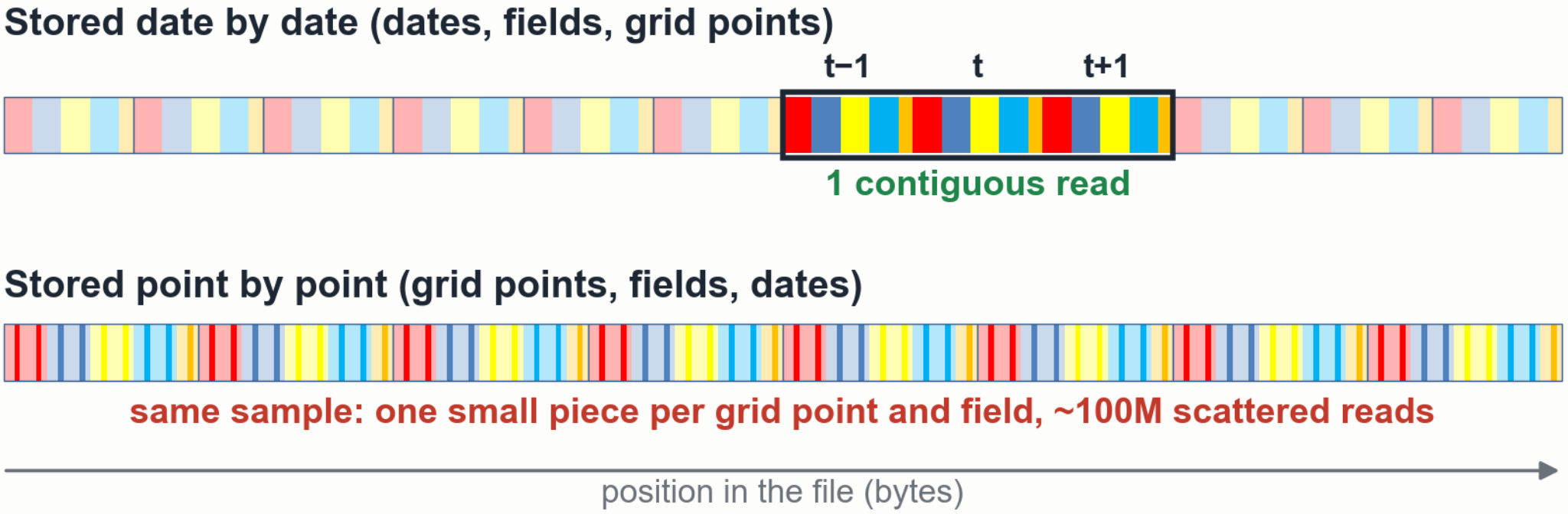
Two dates as input, one as target

In memory: N320 **≈ 600 MB** · o1280 **≈ 9 GB**
On disk, compressed ~2×: N320 **≈ 300 MB** · o1280 **≈ 4.5 GB**

In memory = grid points × ~100 fields × 4 bytes (float32). Compression on disk is about two times. N320 (AIFS Single): 542,080 points. o1280: 6,599,680 points.

On disk, one file is a line

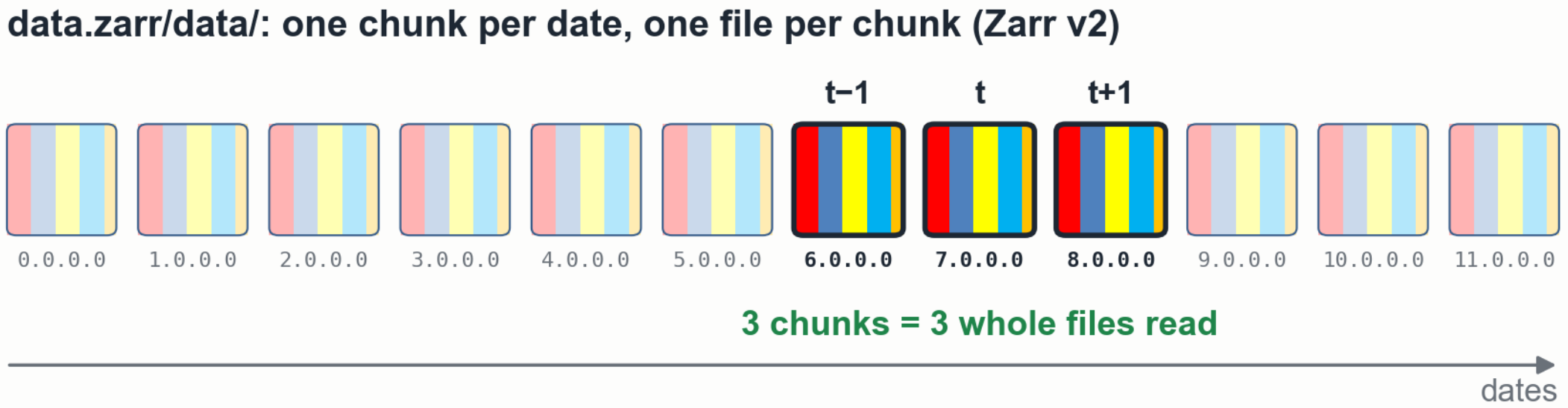
A file is one-dimensional: the order of the dimensions decides which bytes sit next to each other



- **Hard drive:** reading sequentially is the most efficient. One seek, then the disk streams at full speed.
- **But it depends:** SSD, Lustre and S3 care less about order, more about the size and number of requests.

In practice: not one file, but chunks

Anemoi datasets are Zarr: one chunk per date, and in Zarr v2 each chunk is its own file



Nicely efficient: one training sample is 3 chunks, so 3 large reads of whole files. No wasted bytes, few requests.

But it depends on the chunk size: about 100 MB per chunk at N320, 1.5 GB at o1280. Too small: millions of files and requests. Too big: you read bytes you don’t need.

Know your access patterns: transposition

Write the dataset layout according to your reading pattern

Read pattern orthogonal to the pattern on disk?

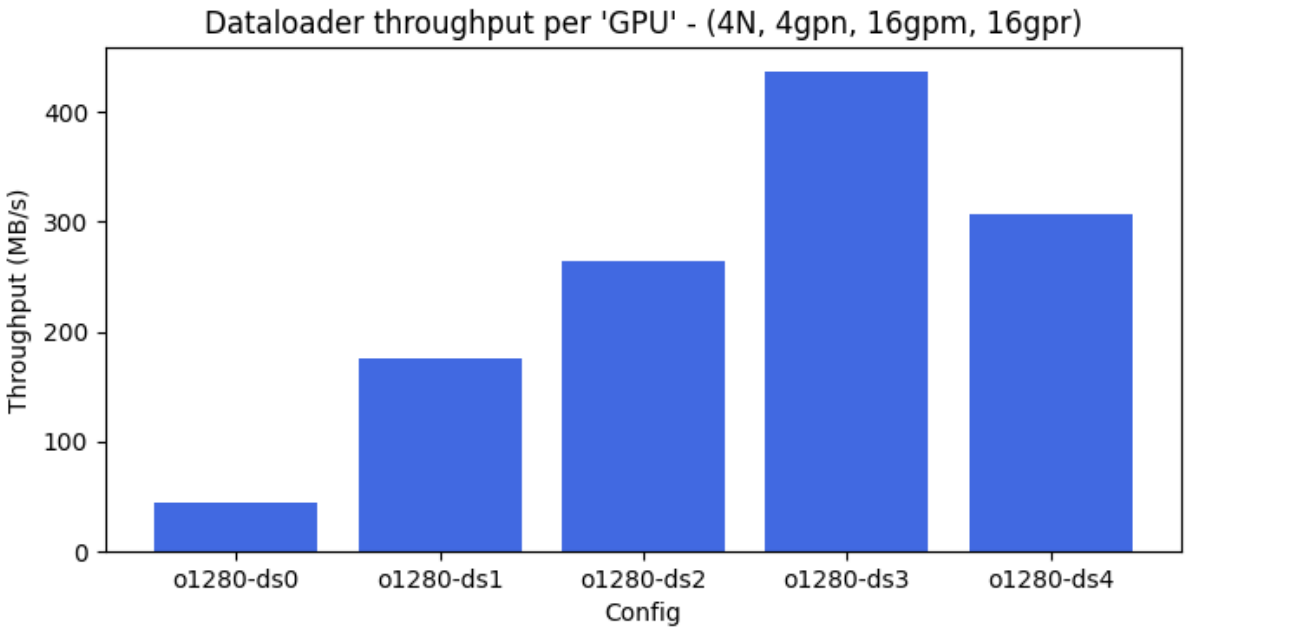
⚠ **No** way to read input data efficiently

✓ **Rewrite** the data to adapt to the read pattern

- You read timeseries? Store the data **by location**.
- You read training samples, all fields for one date? Store the data **by date** (but “it depends”).
- Build a dataset dedicated to your training task, and do the transposition **offline**, once.

Reformatted datasets: a chunk-size experiment

Same idea as transposition: match how the data is stored to how it is read



The experiment

Same o1280 data, rewritten with different chunk sizes (ds0 to ds4). Only the chunking changes, and throughput changes a lot.

Still Zarr v2

These datasets use Zarr v2: one chunk per file.

Next: Zarr v3

Anemoi datasets now supports Zarr v3. More benchmarks to come, using its ability to store several chunks in the same file.

Zarr 2 and Zarr 3?

THE SOFTWARE

Python package zarr

zarr 2

Previously used

zarr 3

Anemoi datasets uses it now (recent update)
Can read and write zarr 2
Reads in parallel (async): faster on S3, not on Lustre or SSD

THE DATA

Zarr format on disk

Format v2

What we still build today

Format v3

Coming soon

WHAT FORMAT V3 BRINGS

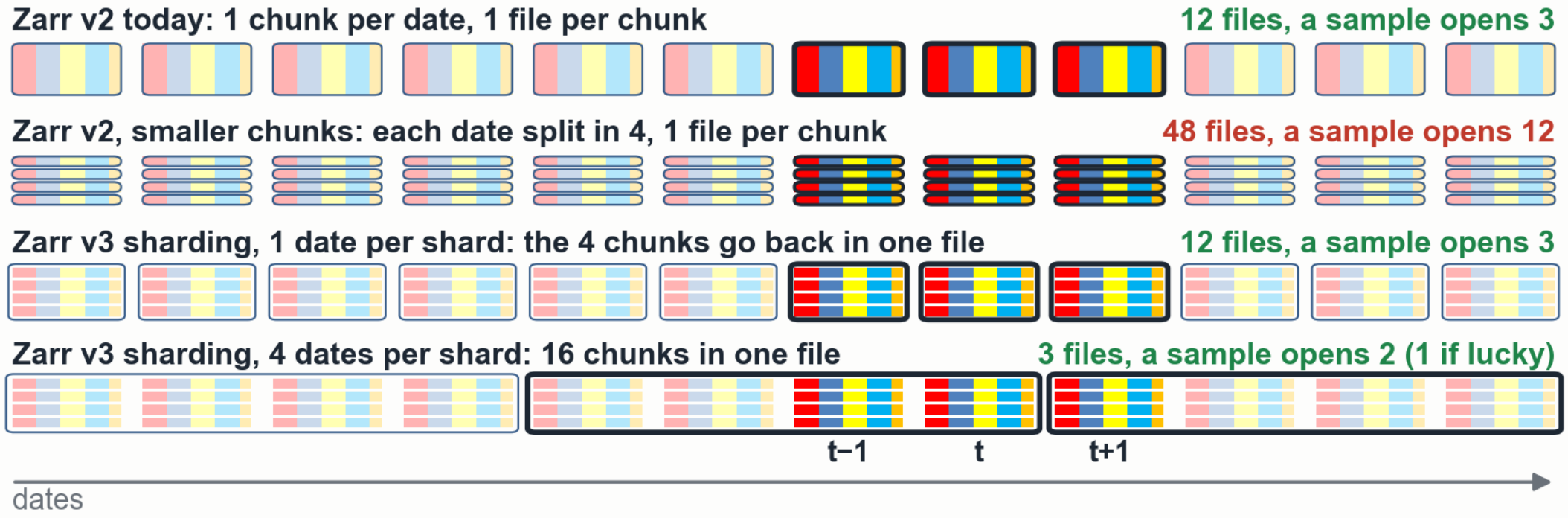
Several chunks in one file (sharding): far fewer inodes used

SOON

Start building datasets in format v3, after giving people time to update their environments
Format v2 stays possible with a config key

Zarr v3 sharding: small chunks, few files

Split a date into smaller chunks, then regroup the chunks into shard files



Best of both: small chunks, so you can read only part of a date, and few files, so far fewer inodes and opens.

Still match the read pattern: t−1, t, t+1 often straddle two shards. Landing in one shard is luck, not design.

How to shard: the order of the chunks inside a shard

A shard is a file, so a line again. One shard here: 4 dates × 4 grid parts (A to D) = 16 chunks

Training on 1 GPU: it reads whole dates t−1, t, t+1

Shard ordered date by date



1 contiguous read

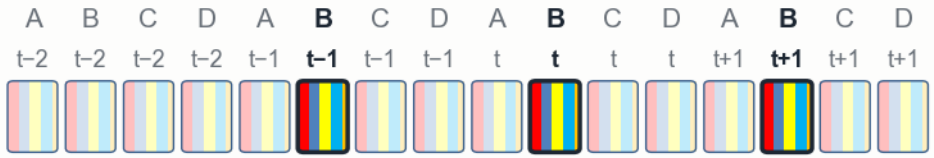
Shard ordered grid part by grid part



4 separate reads

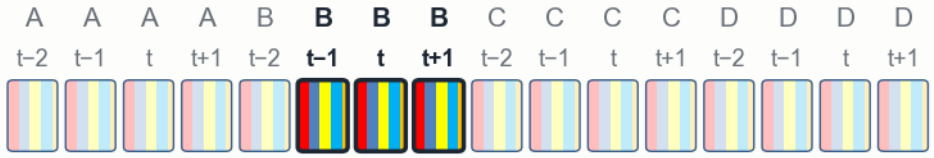
Training on 4 GPUs: each reads its grid part, here B

Shard ordered date by date



3 separate reads

Shard ordered grid part by grid part



1 contiguous read

⚠ **It depends on how many GPUs share a sample:** the best order for 1 GPU is the worst for 4.

✓ Same rule as before: **write the layout for the way you will read it**, down to the order inside the shard.

Know your access patterns: parallel readers

> **Adapt the layout to the read pattern, and the read pattern depends on how many readers run in parallel: GPUs, model groups, dataloader workers**

⚠ Non-contiguous reads may not be too bad

It depends: on the storage, on the size of each piece, on the caches, and on how many readers share the load. Many parallel readers can hide the cost of scattered reads.

WHAT MATTERS

- ✓ **Don’t starve your GPUs**
- ✓ **Be a nice HPC citizen**
- ~~Be as fast as possible~~

04

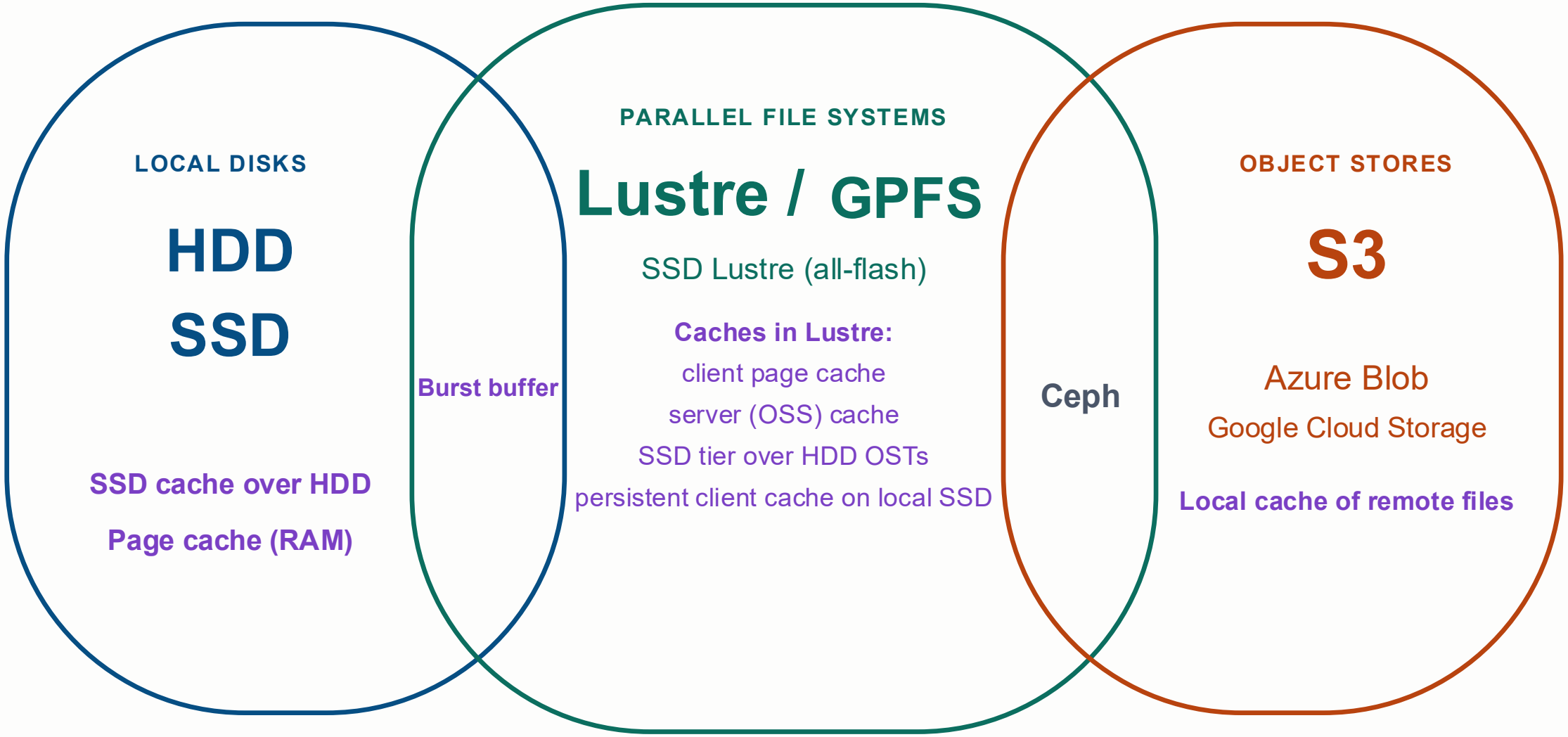
Storage backends

Where the data lives changes how it should be read

ECMWF

Storage: many kinds

Purple = a cache



Local disks: HDD and SSD

HDD: one spinning disk

- ⚠ One moving arm: **one request at a time**
- ✓ Sequential reads are fine
- ⚠ **Non-sequential reads kill it**: every jump is a seek of several milliseconds (about 100–200 random reads per second)

SSD: flash memory

- ✓ **Much faster**: no moving parts
- ✓ Serves many requests in parallel
- ⚠ Jumping around is still not free, **but not as bad** as on an HDD

> On both, read large contiguous blocks: that is why the data layout matters

Shared storage: Lustre and S3

Lustre: parallel file system

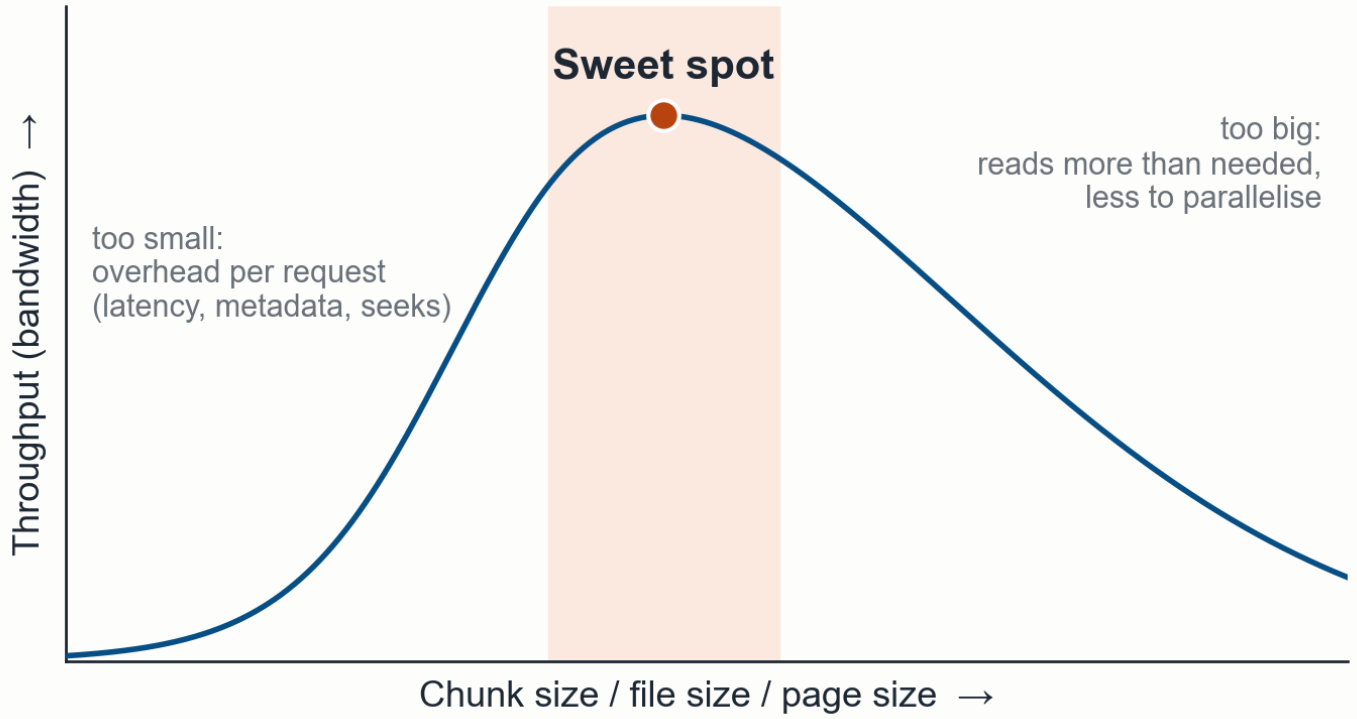
- Data is spread over many **OSTs** (object storage targets), large files can be striped: high bandwidth in parallel
- **Metadata servers** handle every open, stat and listing: many small files overload them
- **Shared with many users**: your speed depends on everyone else's jobs

S3: object store

- Every read is an HTTP request: **high latency** per request
- Throughput comes from **many requests in parallel**

> Fewer, larger files help Lustre (zarr format v3 sharding); parallel reads help S3 (zarr 3 package)

Parallel is not automatically faster



The sweet spot **moves**: it depends on the storage, the data you load, the read pattern, the number of readers, the network… and many more things.

Don’t assume that because you parallelise, it will be faster

- **Find the bottleneck**
- **Do measurements**
- **Do not starve your GPUs**

05

Sharing training datasets

Build on shared Anemoi datasets instead of private copies

ECMWF

Two ways to get your training data

OLD STYLE

Everyone builds their own dataset

- Create a full dataset for your experiment
- Train on it
- You own it: one copy per scientist.

ANEMOI

Build on shared Anemoi datasets

- Start from datasets that are already built and shared
- Create only a tiny dataset with the extra data you need
- Combine them when loading; share and reuse

Why sharing datasets pays off



Disk space

One shared copy instead of one per scientist. Your own dataset stays tiny.



Bandwidth

Data read by everyone stays warm in the caches, so it is fetched once, not again and again.



Moving between HPCs

Common datasets are transferred once and shared. Only your small dataset needs to move.

A dataset name is a guarantee

Once built, a dataset never changes

Same name = same data, everywhere, forever. Different data (e.g. with different variables or dates) gets a new name (a new version).

Share

Everyone who uses the name trains on exactly the same data. Copies on other HPCs and in caches are identical: nothing to check again.

Cite and reproduce

Cite the dataset in papers and model cards. Give it a **DOI**, so others can find it and rerun the training.

A different workflow, but not so different.

```
from anemoi.datasets import open_dataset

ds = open_dataset(join=[
"shared-dataset-name", # already built, shared
"my-tiny-dataset", # only the extra data you need
])
```

open_dataset: everything composes

Operations take datasets and return datasets, so they chain without limit. Nothing is read until you index

Step by step

```
ds = open_dataset(”extra-variables-dataset”
ds = open_dataset(select = [”tp”, “cp”])
ds = open_dataset(join=[ds, “shared-dataset-name”])
ds = open_dataset(ds, start=2000, end=2020)
ds = open_dataset(ds, frequency="12h")
```

Each call wraps the previous one: open, then subset, then select, then change the frequency.

The same, as yaml

```
dataset:
  join:
    - shared-dataset-name
    - dataset: extra-variables-dataset
      select: [tp, cp]
  start: 2000
  end: 2020
  frequency: 12
```

A whole pipeline in a YAML file: the training config hands it straight to **open_dataset**.

Selecting variables: select, drop, reorder, rename, rescale

Filter the second dimension, the variables

Keep or remove

```
open_dataset(dataset, select=["2t", "tp"])
# list: this order
open_dataset(dataset, select={"2t", "tp"})
# set: original order
open_dataset(dataset, drop=["10u", "10v"])
```

Rename, reorder, convert units

```
open_dataset(dataset, rename={"2t": "t2m"})

open_dataset(dataset, reorder={
  "2t": 0, "msl": 1, "sp": 2, "10u": 3, "10v": 4})

open_dataset(dataset, rescale={
  "2t": ("K", "degC"), "tp": ("m", "mm")})
```

rename is useful when you join datasets: it stops a variable of one dataset from overriding the others.
rescale also rescales the statistics.

Subsetting dates and grids: start, end, frequency, area

Filter the first dimension, the dates, and the last one, the grid points

Dates

```
open_dataset(dataset, start=1980)
open_dataset(dataset, end="2023-06")
open_dataset(dataset, frequency="6h")
open_dataset(dataset, interpolate_frequency="10m")
open_dataset(dataset,
  extend_start="2019-01-01", fill_missing_dates="nans")
```

start="2020" means 1 January, **end="2020"** means the last date of 2020. A new **frequency** must be a multiple of the old one.

Examples from the anemoi-datasets documentation: *Using datasets*

Grid points

```
open_dataset(dataset, area=(60, -50, 20, 0))
# north, west, south, east

open_dataset(dataset, thinning=N,
  method="every-nth")
```

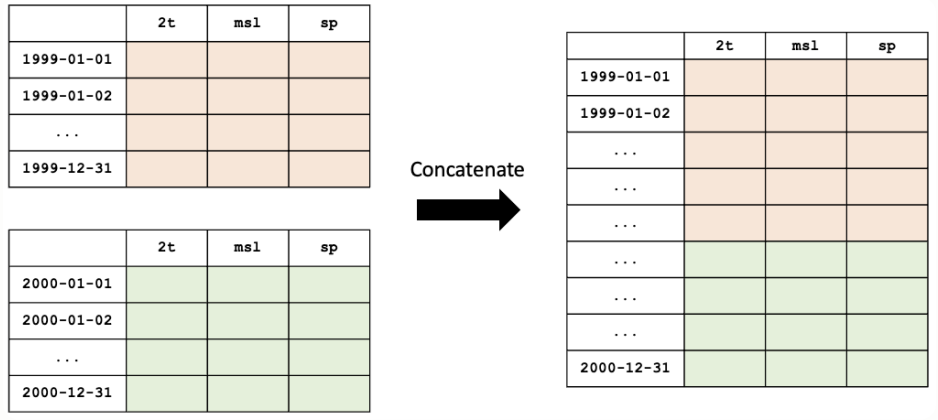
area keeps a box. **thinning** keeps one point in N, on regular lat/lon grids.

Combining datasets: concat and join

A “virtual” dataset made of others, loaded lazily from the originals

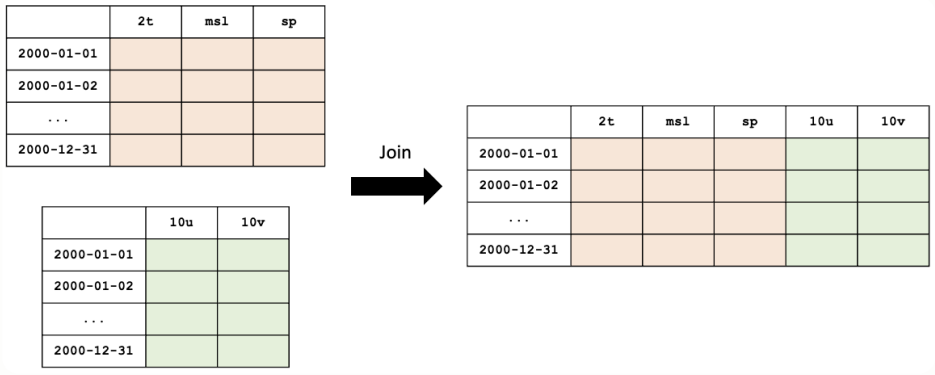
concat: more dates

```
open_dataset(concat=[
  "dataset-1979-2000",
  "dataset-2001-2022",
])
```



join: more variables

```
open_dataset(join=[
  "dataset1-1979-2022",
  "dataset2-1979-2022",
])
```



A plain list does either, automatically: same variables and consecutive dates → concat; same dates → join.

Combining datasets: ensembles, grids, cutout

Combine along the other dimensions: members and grid points

Along members and grids

```
open_dataset(ensemble=[dataset1, dataset2])
# more ensemble members

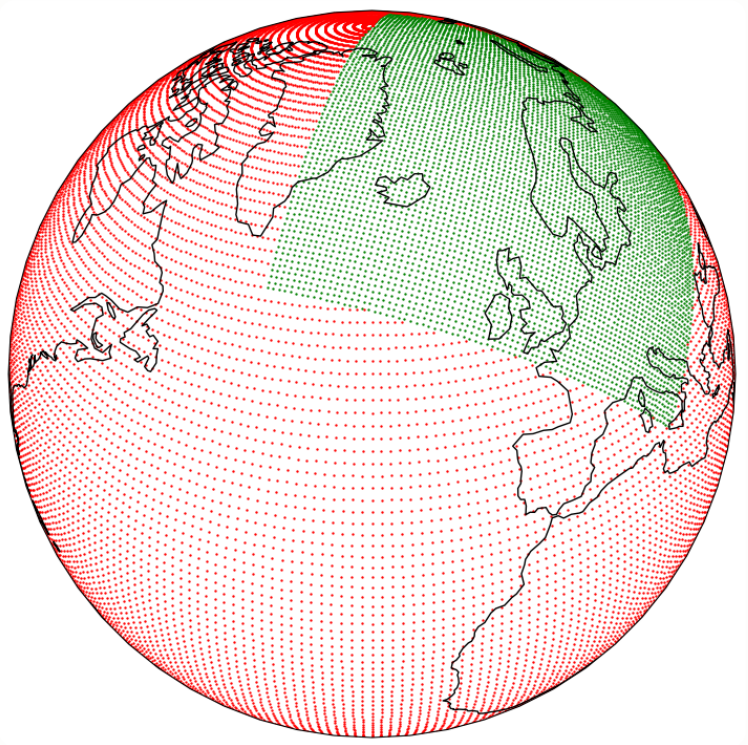
open_dataset(grids=[dataset1, dataset2])
# grid points one after the other

open_dataset(cutout=[lam_dataset, global_dataset])
# the regional grid replaces the global one

open_dataset(dataset, statistics=other_dataset)
# normalise with another dataset's statistics
```

Every element can itself be a dictionary with its own keywords: **{"dataset": …, "end": 2010}**.

cutout



06

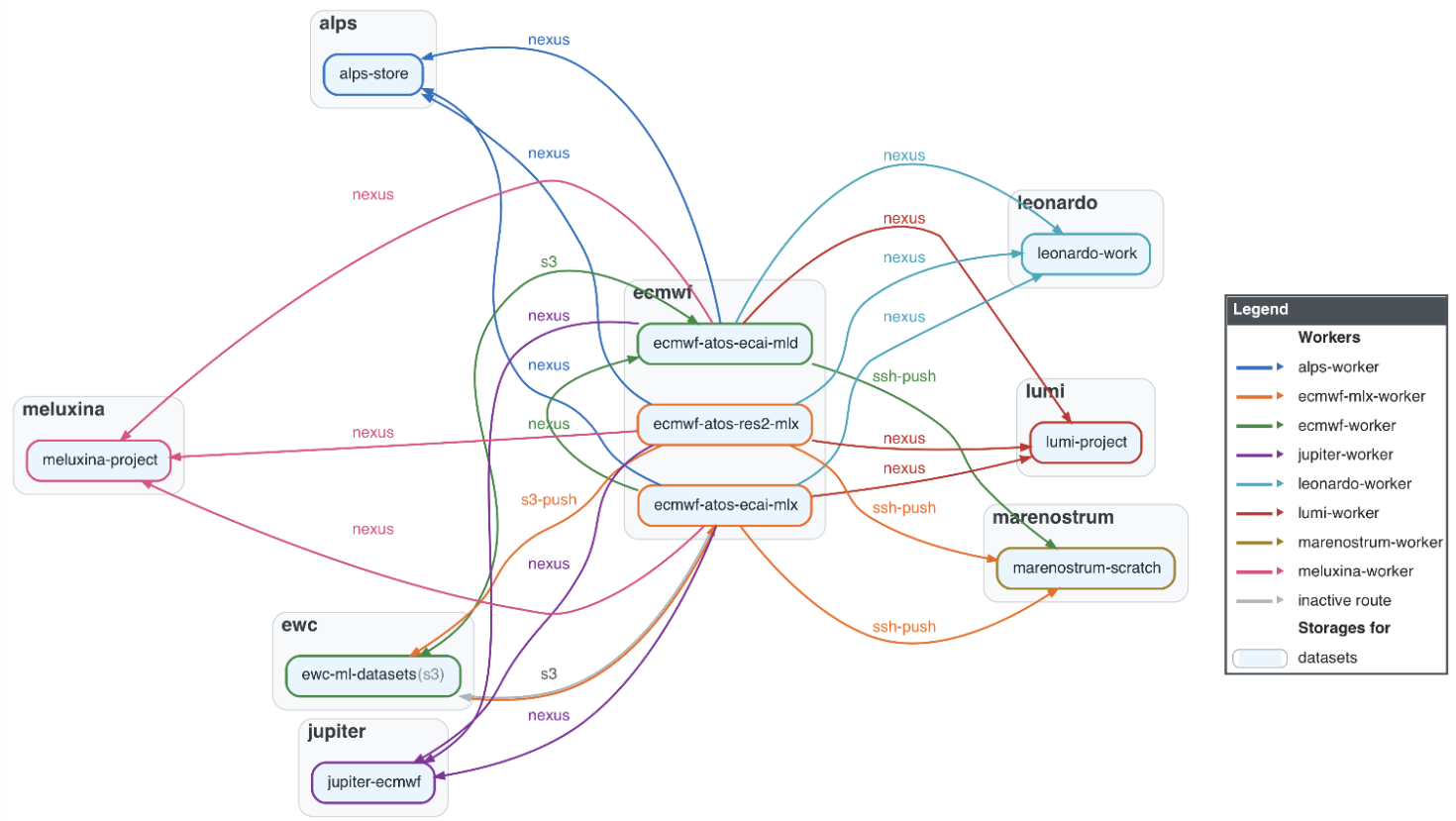
Moving datasets between HPCs

Copy once, the same name everywhere

ECMWF

An infrastructure to move datasets between HPCs

Datasets are built at ECMWF, then copied by workers to the storage of each HPC



It relies on: a name is a guarantee

Datasets are immutable: a copy is identical to the original, or not there yet. Nothing to sync, no versions to reconcile

8 sites today

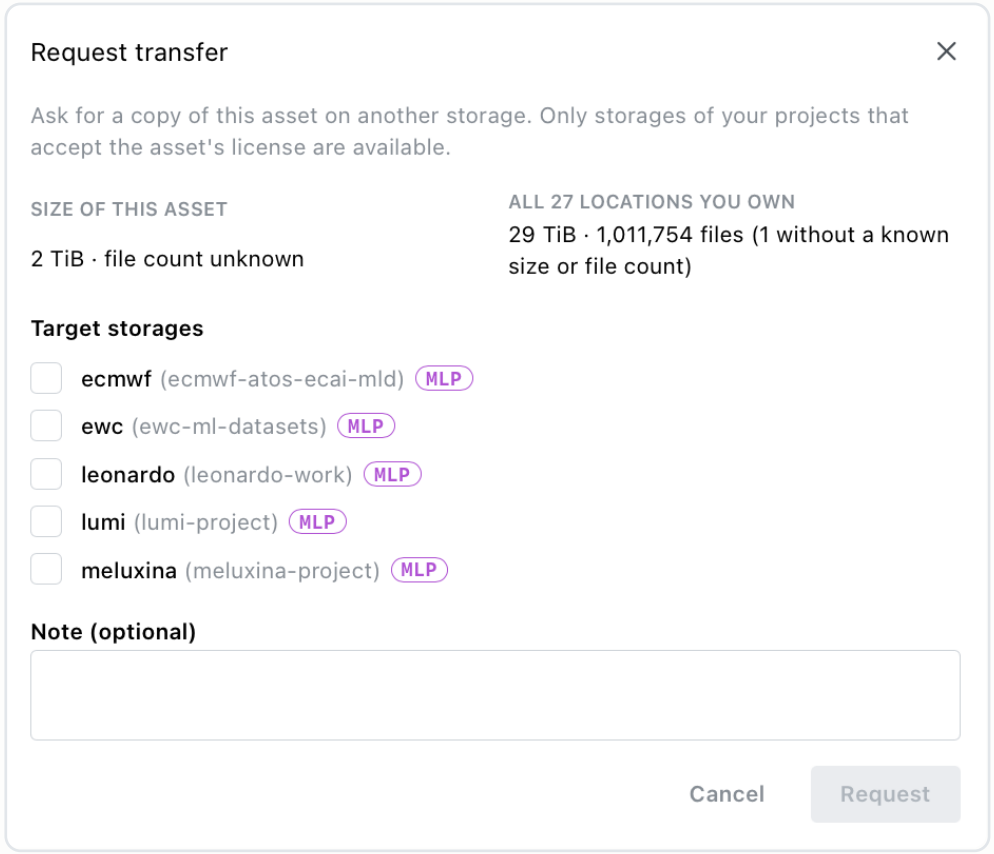
ECMWF, ALPS, JUPITER, Leonardo, LUMI, MareNostrum, MeluXina and the EWC

Same name, same data, everywhere

Your recipe and your **open_dataset** call move with you from one HPC to the next

Asking for a copy: request, approve, transfer

Nobody copies datasets by hand: you ask, the storage owner approves, a worker copies



1 Request

On the dataset's page, tick the target storages. Only storages of your projects that accept the dataset's licence are offered

2 Approve

An admin, or the owner of the managed storage, approves: it is their quota, bytes and inodes

3 Transfer

A worker copies the dataset and registers the new copy: the location goes from **pending** to **available**

Every copy is registered

The catalogue knows which dataset is on which storage, and how full each storage is. Snapshot, November 2026

STORAGE	SITE	BYTES USED / QUOTA	INODES USED / QUOTA	COPIES
alps-store	ALPS	74.1 TiB / 1000 TiB	1.1M / 50.0M	14
ecmwf-atos-ecai-mlx	ECMWF	502.4 TiB / 2.3 PiB	12.0M	804
ecmwf-atos-res2-mlx	ECMWF	707.1 TiB / 847.7 TiB	16.9M	286
ewc-ml-datasets (S3)	EWC	435.4 TiB / 700 TiB	12.8M / 20.0M	261
jupiter-ecmwf	JUPITER	15.8 PiB / 19.5 PiB	52.9M / 70.0M	97
leonardo-work	Leonardo	915.5 TiB / 2.5 PiB	13.5M	136
lumi-project	LUMI	–	–	27
marenostrum-scratch	MareNostrum	5.5 PiB / 6.5 PiB	714.6M	116
meluxina-project	MeluXina	283.0 TiB / 414.2 TiB	10.4M / 15.0M	29

One name, many copies

The catalogue records every copy of every dataset, on every storage

Two quotas: bytes and inodes

On a parallel file system you can run out of files before you run out of space

Not only datasets

The same storages hold model weights and other artefacts

07

Other places to access Anemoi datasets

Open Anemoi datasets, ready where the GPUs are

ECMWF

Open Anemoi datasets: one today, more coming

Public dataset in the catalogue today is ERA5

AVAILABLE NOW

`era5-o96-1979-2023-6h-v8`

ERA5 on the O96 grid (about 1°), every 6 hours, 1979 to 2023. About 0.5 TB. Licence CC-BY-4.0.

```
pip install "anemoi-datasets[remote]" aiohttp
# only if you get CERTIFICATE_VERIFY_FAILED:
export SSL_CERT_FILE=$(python -m certifi)

anemoi-datasets copy \
  https://data.ecmwf.int/anemoi-datasets/era5-o96-1979-2023-6h-v8.zarr \
  era5-o96-1979-2023-6h-v8.zarr
```

Copy it to your HPC first, then train on the local copy. Use **--resume** if it stops.

COMING

More open datasets

- including **observations**
- available on the **AI Factories**, close to the GPUs

EU AI Factories

AI-optimised supercomputers selected by the EuroHPC Joint Undertaking: 19 sites between December 2024 and October 2025

LUMI AI Factory

Kajaani, Finland

BSC AI Factory

Barcelona, Spain

> **ECMWF is working with them on a collection of 9 Anemoi datasets that can be accessed by all AI factory users.**

09

Wrap-up

The whole talk, in one slide

ECMWF

Effective data loading, in one slide

01	**Why data movement matters**	Compute grew faster than memory and bandwidth: loading the data is often the bottleneck
02	**Zarr**	Anemoi datasets are Zarr stores. Use the public API: the recipe with **create**, and **open_dataset()**
03	**Data layout**	Store data the way it is read: chunks, shards and their order follow the access pattern
04	**Storage backends**	Each backend wants to be read differently. Parallel is not automatically faster: measure
05	**Sharing training datasets**	Build on shared, immutable datasets and compose them with **open_dataset**
06	**Moving datasets**	Immutable, named datasets are copied once to each HPC by a shared infrastructure
07	**AI Factories**	Open Anemoi datasets where the GPUs are; more coming, observations included
08	**GPU data loading**	Fill the GPU and the CPU, and don't starve your GPU

> **Share the data, read it the way it is stored, and keep the GPU fed**