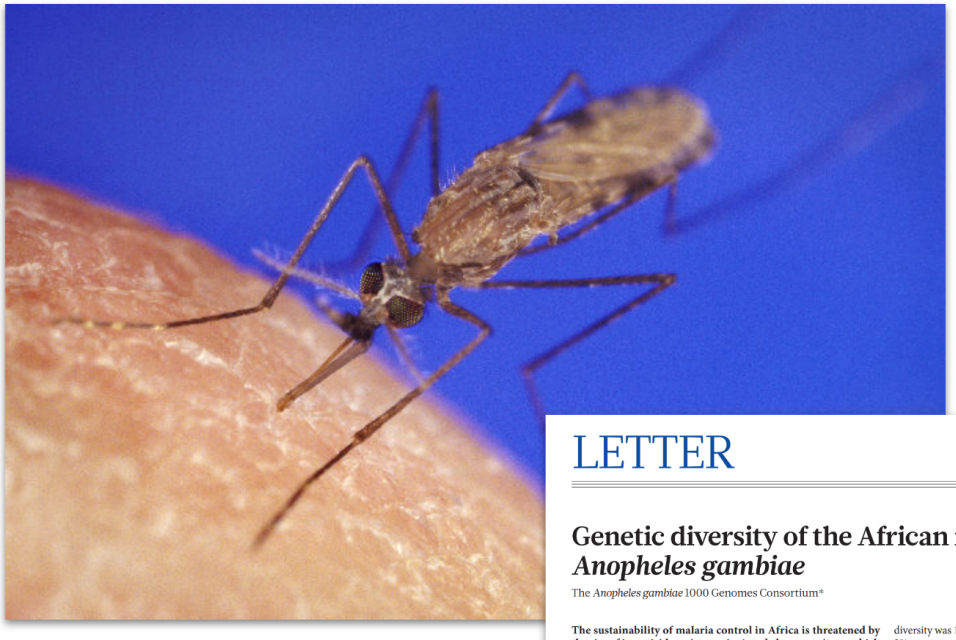
Zarr - scalable storage of tensor data for parallel and distributed computing

Alistair Miles (@alimanfoo) - SciPy 2019

These slides: https://zarr-developers.github.io/slides/scipy-2019.html

Malaria prevention in Africa



LETTER

doi:10.1038/nature24995

Genetic diversity of the African malaria vector *Anopheles gambiae*

The *Anopheles gambiae* 1000 Genomes Consortium*

The sustainability of malaria control in Africa is threatened by the rise of insecticide resistance in *Anopheles* mosquitoes, which transmit the disease[1]. To gain a deeper understanding of how mosquito populations are evolving, here we sequenced the genomes of 765 specimens of *Anopheles gambiae* and *Anopheles coluzzii* sampled from 15 locations across Africa, and identified over 50 million single nucleotide polymorphisms within the accessible genome. These data revealed complex population structure and patterns of gene flow, with evidence of ancient expansions, recent bottlenecks, and local variation in effective population size. Strong signals of recent selection were observed in insecticide-resistance genes, with several sweeps spreading over large geographical distances and between species. The design of new tools for mosquito control using gene-drive systems will need to take account of high levels of genetic diversity in natural mosquito populations.

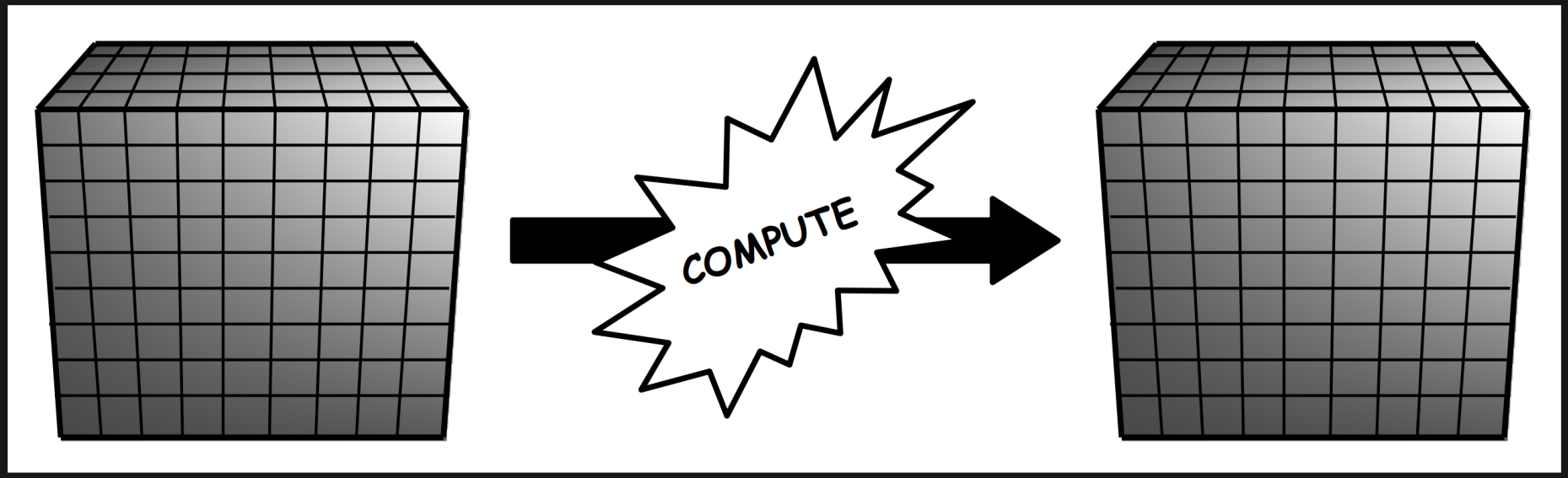
Blood-sucking mosquitoes of the *An. gambiae* species complex are

diversity was 1.5% on average (Extended Data Fig. 3b) and more than 3% at synonymous coding sites (Extended Data Fig. 3c), confirming that these are among the most genetically diverse eukaryotic species[9].

High levels of natural diversity have practical implications for the development of gene-drive technologies for mosquito control[10]. CRISPR–Cas9 gene drives can be designed to edit a specific gene and confer a phenotype such as female sterility, which could suppress mosquito populations and thereby reduce disease transmission. However, naturally occurring polymorphisms within the approximately 21-base-pair (bp) Cas9 target site could prevent target recognition, and thus undermine gene-drive efficacy in the field. We found viable Cas9 targets in 11,625 protein-coding genes, but only 5,474 genes remained after excluding target sites with nucleotide variation in any of the 765 genomes sequenced here (Extended Data Fig. 3d; Supplementary Information 5). Resistance to gene drive could be countered by designing constructs that target multiple sites within the

Motivation: Why Zarr?

Problem statement



There is some computation we want to perform.

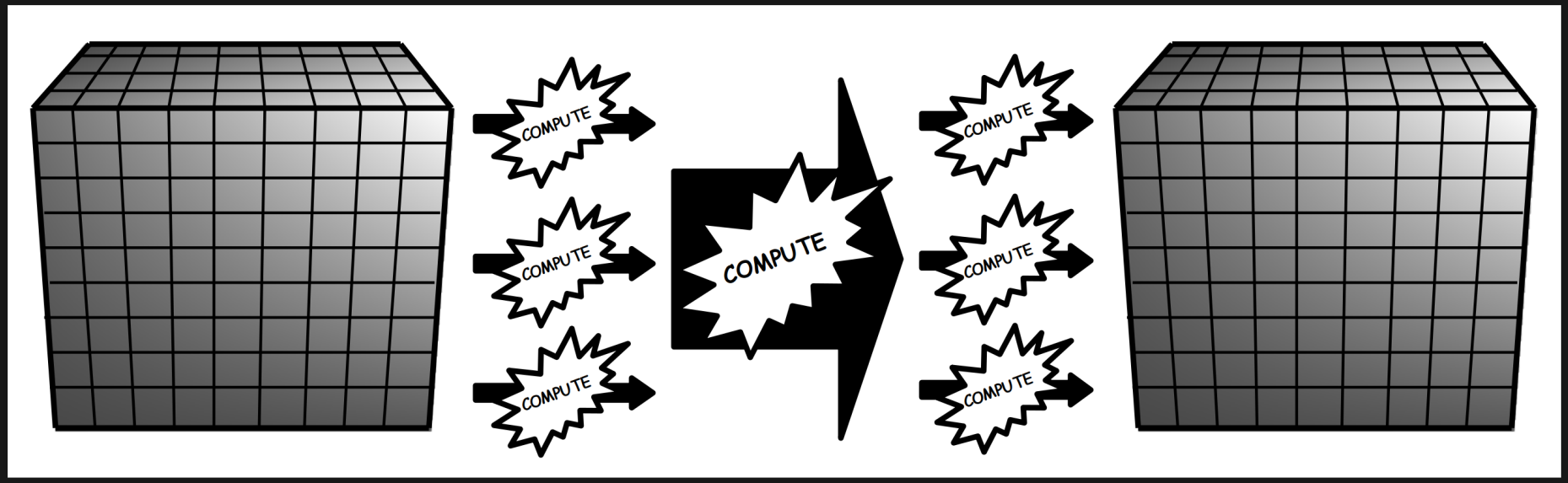
Inputs and outputs are multidimensional arrays (a.k.a. tensors).

5 key features...

(1) Larger than memory

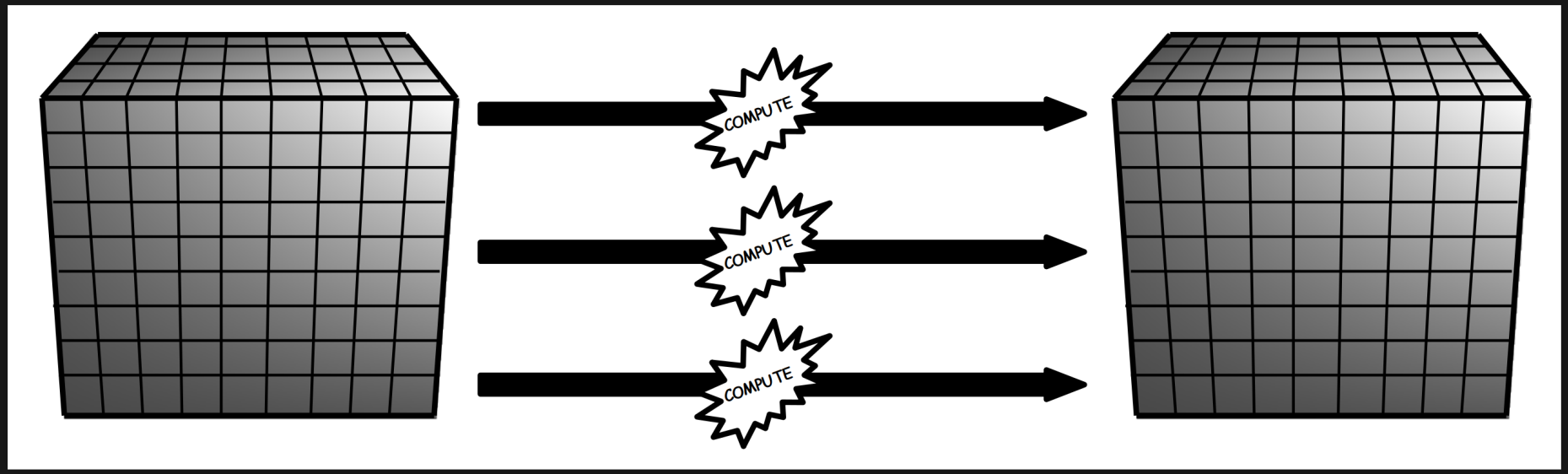
Input and/or output tensors are too big to fit comfortably in main memory.

(2) Computation can be parallelised



At least some part of the computation can be parallelised by processing data in chunks.

E.g., embarassingly parallel



(3) I/O is the bottleneck

Computational complexity is moderate → significant amount of time is spent in reading and/or writing data.

N.B., bottleneck may be due to (a) limited I/O bandwidth, (b) I/O is not parallel.

(4) Data are compressible

- Compression is a very active area of innovation.
- Modern compressors achieve good compression ratios with very high speed.
- Compression can increase effective I/O bandwidth, sometimes dramatically.

(5) Speed matters

- Rich datasets → exploratory science → interactive analysis → many rounds of summarise, visualise, hypothesise, model, test, repeat.
- E.g., genome sequencing.
  - Now feasible to sequence genomes from 100,000s of individuals and compare them.
  - Each genome is a complete molecular blueprint for an organism → can investigate many different molecular pathways and processes.
  - Each genome is a history book handed down through the ages, with each generation making its mark → can look back in time and infer major demographic and evolutionary events in the history of populations and species.

Problem: key features

0. Inputs and outputs are tensors.
1. Data are larger than memory.
2. Computation can be parallelised.
3. I/O is the bottleneck.
4. Data are compressible.
5. Speed matters.

Solution

1. Chunked, parallel tensor computing framework.
2. Chunked, parallel tensor storage library.

Align the chunks!

Parallel computing framework for chunked tensors.

```
import dask.array as da

a = ...  # what goes here?
x = da.from_array(a)
y = (x - x.mean(axis=1)) / x.std(axis=1)
u, s, v = da.linalg.svd_compressed(y, 20)
u = u.compute()
```

- Write code using a numpy-like API.
- Parallel execution on local workstation, HPC cluster, Kubernetes cluster, ...

- Scale up ocean / atmosphere / land / climate science.
- Aim to handle petabyte-scale datasets on HPC and cloud platforms.
- Using Dask.
- Needed a tensor storage solution.
- Interested to use cloud object stores: Amazon S3, Azure Blob Storage, Google Cloud Storage, ...

Tensor storage: prior art

HDF5 (h5py)

- Store tensors ("datasets").
- Divide data into regular chunks.
- Chunks are compressed.
- Group tensors into a hierarchy.
- Smooth integration with NumPy...

```
import h5py
x = h5py.File('example.h5')['x']
# read 1000 rows into numpy array
y = x[:1000]
```

HDF5 - limitations

- No thread-based parallelism.
- Cannot do parallel writes with compression.
- Not easy to plug in a new compressor.
- No support for cloud object stores (but see Kita).

See also moving away from HDF5 by Cyrille Rossant.

bcolz

- Developed by Francesc Alted.
- Chunked storage, primarily intended for storing 1D arrays (table columns), but can also store tensors.
- Implementation is simple (in a good way).
- Data format on disk is simple - one file for metadata, one file for each chunk.
- Showcase for the Blosc compressor.

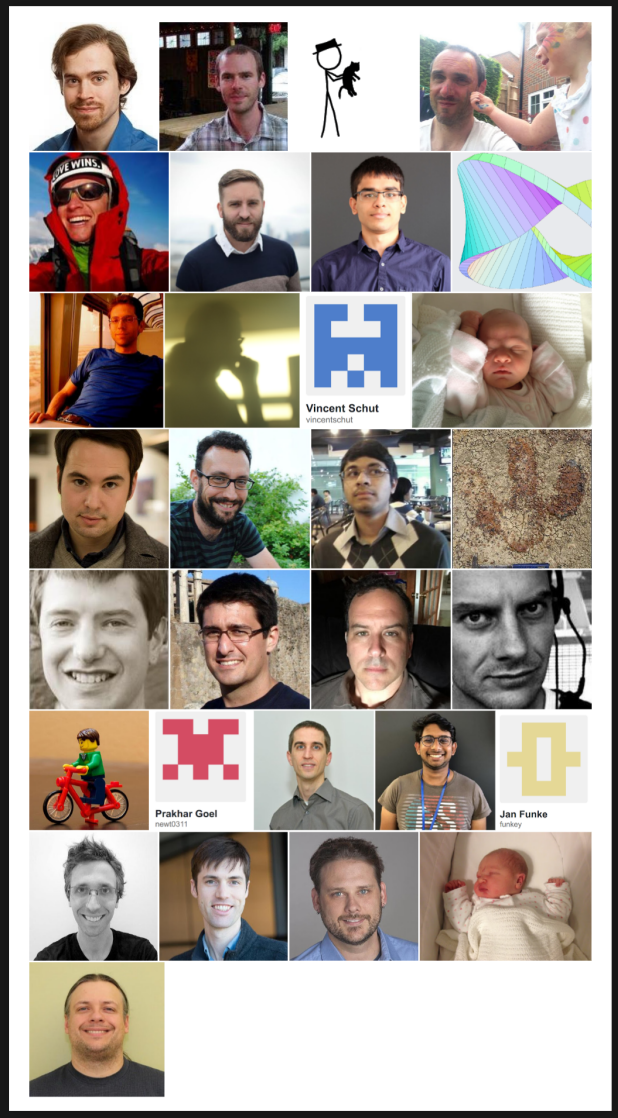
bcolz - limitations

- Chunking in 1 dimension only.
- No support for cloud object stores.

How hard could it be ...

... to implement a chunked storage library for tensor data that supported parallel reads, parallel writes, was easy to plug in new compressors, and easy to plug in different storage systems like cloud object stores?

<montage/>



3 years, 1,107 commits, 39 releases, 259 issues, 165 PRs, and at least 2 babies later

...

Zarr Python

```
$ pip install zarr
```

```
$ conda install -c conda-forge zarr
```

```
>>> import zarr
>>> zarr.__version__
'2.3.2'
```

Conceptual model based on HDF5

- Multiple arrays (a.k.a. datasets) can be created and organised into a hierarchy of groups.
- Each array is divided into regular shaped chunks.
- Each chunk is compressed before storage.

Creating a hierarchy

```
>>> store = zarr.DirectoryStore('example.zarr')
>>> root = zarr.group(store)
>>> root
<zarr.hierarchy.Group '/'>
```

Using DirectoryStore the data will be stored in a directory on the local file system.

Creating an array

```
>>> hello = root.zeros('hello',
...                    shape=(10000, 10000),
...                    chunks=(1000, 1000),
...                    dtype='<i4')
>>> hello
<zarr.core.Array '/hello' (10000, 10000) int32>
```

- Creates a 2-dimensional array of 32-bit integers with 10,000 rows and 10,000 columns.
- Divided into chunks where each chunk has 1,000 rows and 1,000 columns.
- There will be 100 chunks in total, arranged in a 10x10 grid.

Creating an array (h5py-style API)

```
>>> hello = root.create_dataset('hello',
...                             shape=(10000, 10000),
...                             chunks=(1000, 1000),
...                             dtype='<i4')
>>> hello
<zarr.core.Array '/hello' (10000, 10000) int32>
```

Creating an array (big)

```
>>> big = root.zeros('big',
...                  shape=(100_000_000, 100_000_000),
...                  chunks=(10_000, 10_000),
...                  dtype='i4')
>>> big
<zarr.core.Array '/big' (100000000, 100000000) int32>
```

Creating an array (big)

```
>>> big.info
Name               : /big
Type               : zarr.core.Array
Data type          : int32
Shape              : (100000000, 100000000)
Chunk shape        : (10000, 10000)
Order              : C
Read-only          : False
Compressor         : Blosc(cname='lz4', clevel=5, shuffle=SHUFFLE, b
Store type         : zarr.storage.DirectoryStore
No. bytes          : 40000000000000000 (35.5P)
No. bytes stored   : 355
Storage ratio      : 112676056338028.2
Chunks initialized : 0/100000000
```

- That's a 35 petabyte array.
- N.B., chunks are initialized on write.

Writing data into an array

```
>>> big[0, 0:20000] = np.arange(20000)
>>> big[0:20000, 0] = np.arange(20000)
```

- Same API as writing into numpy array or h5py dataset.

Reading data from an array

```
>>> big[0:1000, 0:1000]
array([[  0,   1,   2, ..., 997, 998, 999],
       [  1,   0,   0, ...,   0,   0,   0],
       [  2,   0,   0, ...,   0,   0,   0],
       ...,
       [997,   0,   0, ...,   0,   0,   0],
       [998,   0,   0, ...,   0,   0,   0],
       [999,   0,   0, ...,   0,   0,   0]], dtype=int32)
```

- Same API as slicing a numpy array or reading from an h5py dataset.

Chunks are initialized on write

```
>>> big.info
Name                : /big
Type                : zarr.core.Array
Data type           : int32
Shape               : (100000000, 100000000)
Chunk shape         : (10000, 10000)
Order               : C
Read-only           : False
Compressor          : Blosc(cname='lz4', clevel=5, shuffle=SHUFFLE, b
Store type          : zarr.storage.DirectoryStore
No. bytes           : 40000000000000000 (35.5P)
No. bytes stored    : 5171386 (4.9M)
Storage ratio       : 7734870303.6
Chunks initialized : 3/100000000
```

Files on disk

```
$ tree -a example.zarr
example.zarr
├── big
│   ├── 0.0
│   ├── 0.1
│   ├── 1.0
│   └── .zarray
├── hello
│   └── .zarray
└── .zgroup

2 directories, 6 files
```

Array metadata

```
$ cat example.zarr/big/.zarray
{
    "chunks": [
        10000,
        10000
    ],
    "compressor": {
        "blocksize": 0,
        "clevel": 5,
        "cname": "lz4",
        "id": "blosc",
        "shuffle": 1
    },
    "dtype": "<i4",
    "fill_value": 0,
    "filters": null,
    "order": "C",
    "shape": [
        100000000,
        100000000
    ],
    "zarr_format": 2
}
```

Reading unwritten regions

```
>>> big[-1000:, -1000:]
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]], dtype=int32)
```

- No data on disk, fill value is used (in this case zero).

Reading the whole array

```
>>> big[:]
MemoryError
```

- Read the whole array into memory (if you can!)

Pluggable storage

`zarr.DirectoryStore`, `zarr.ZipStore`,
`zarr.DBMStore`, `zarr.LMDBStore`, `zarr.SQLiteStore`,
`zarr.MongoDBStore`, `zarr.RedisStore`,
`zarr.ABSStore`, `s3fs.S3Map`, `gcsfs.GCSMap`, …

DirectoryStore

```
>>> store = zarr.DirectoryStore('example.zarr')
>>> root = zarr.group(store)
>>> big = root['big']
>>> big
<zarr.core.Array '/big' (100000000, 100000000) int32>
```

DirectoryStore (reminder)

```
$ tree -a example.zarr
example.zarr
├── big
│   ├── 0.0
│   ├── 0.1
│   ├── 1.0
│   └── .zarray
├── hello
│   └── .zarray
└── .zgroup

2 directories, 6 files
```

ZipStore

```
$ cd example.zarr && zip -r0 ../example.zip ./*
```

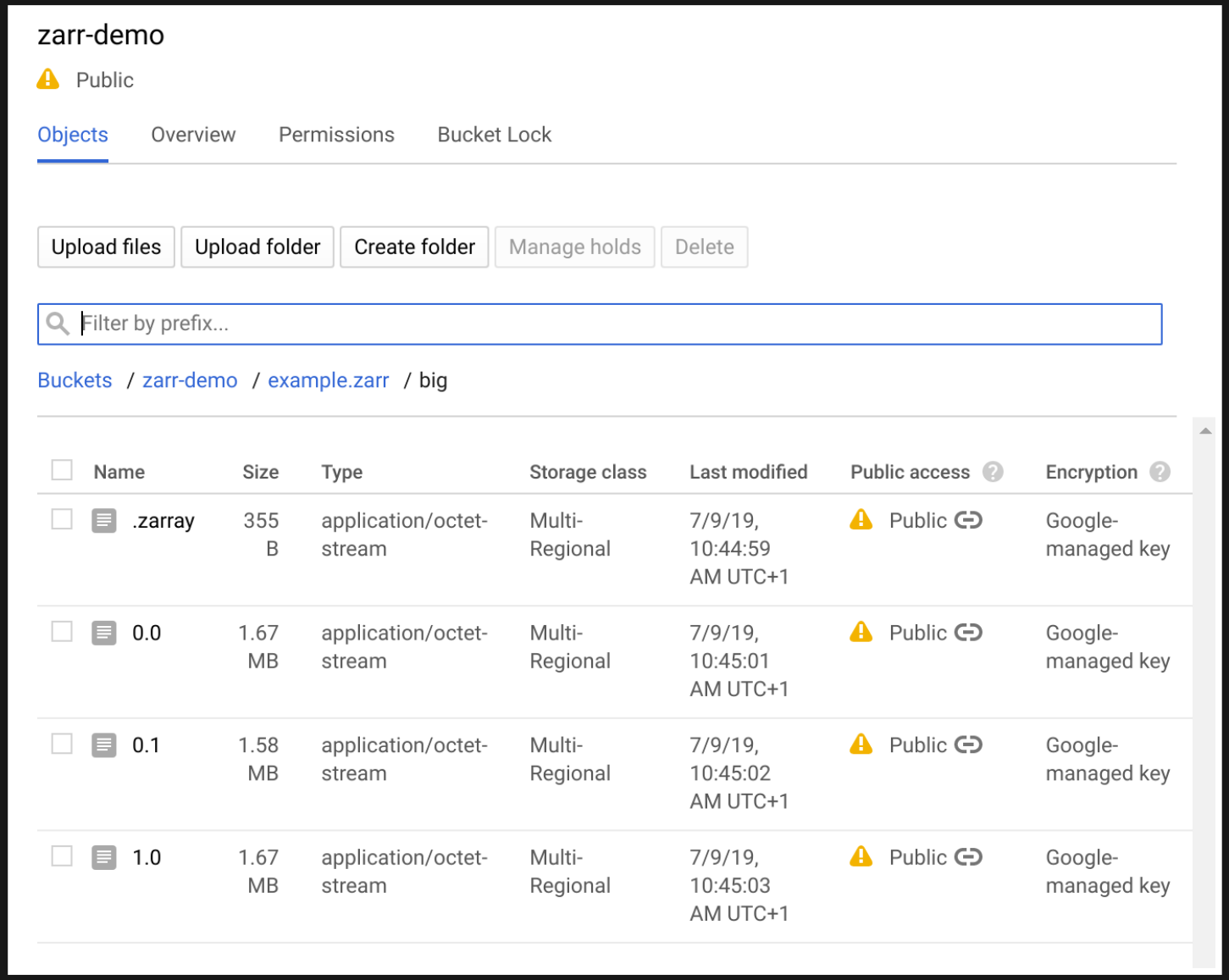
```
>>> store = zarr.ZipStore('example.zip')
>>> root = zarr.group(store)
>>> big = root['big']
>>> big
<zarr.core.Array '/big' (100000000, 100000000) int32>
```

Google cloud storage (via gcsfs)

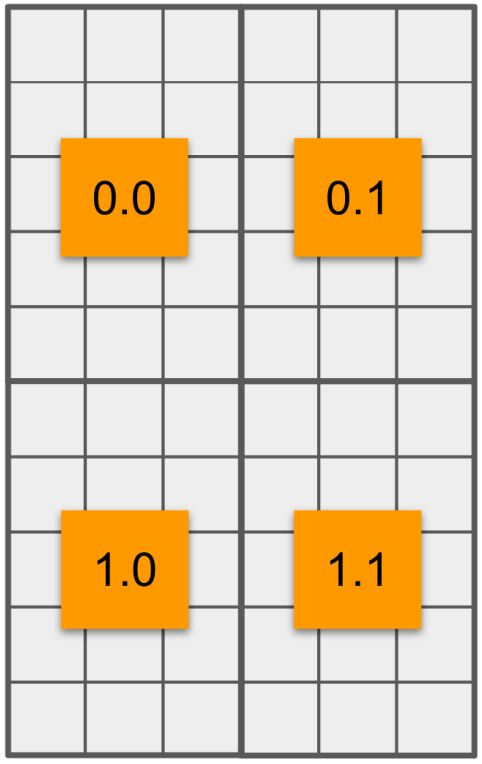
```
$ gsutil config
$ gsutil rsync -ru example.zarr/ gs://zarr-demo/example.zarr/
```

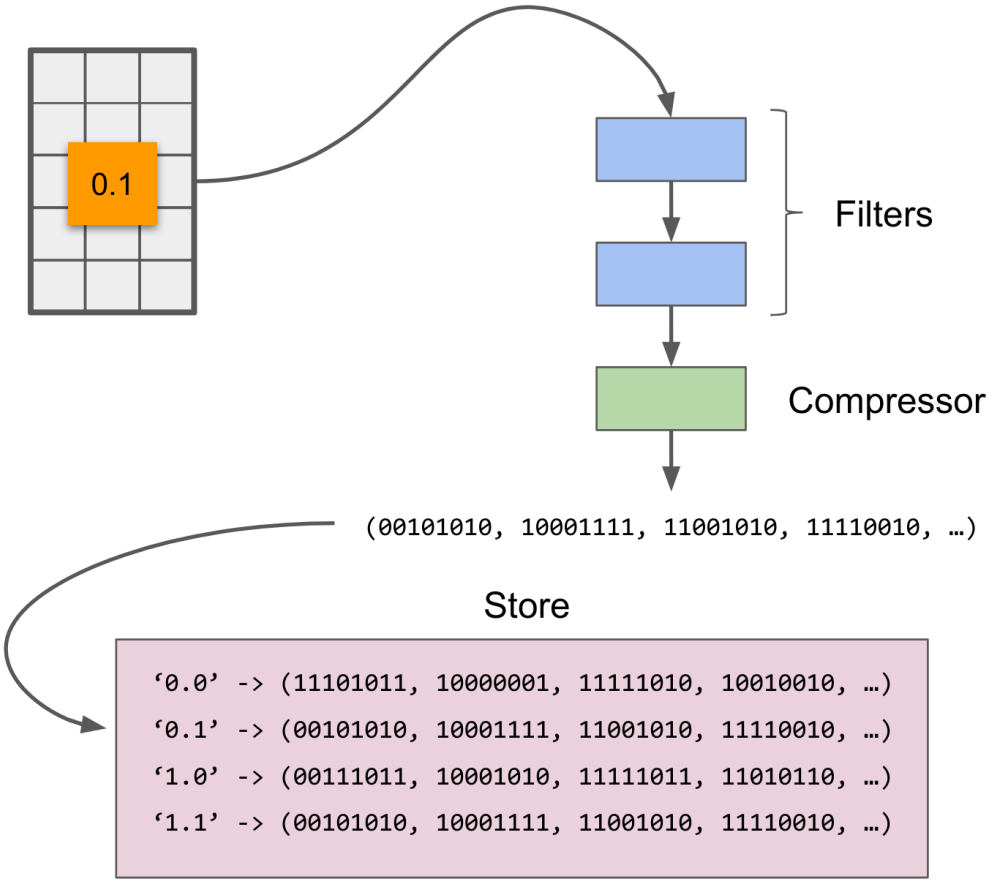
```
>>> import gcsfs
>>> gcs = gcsfs.GCSFileSystem(token='anon', access='read_only')
>>> store = gcsfs.GCSMap('zarr-demo/example.zarr', gcs=gcs, check=Fa
>>> root = zarr.group(store)
>>> big = root['big']
>>> big
<zarr.core.Array '/big' (100000000, 100000000) int32>
```

Google cloud storage



E.g., array with shape (10, 6) and chunk shape (5, 3) has 4 chunks in a 2 by 2 chunk grid, with chunks identified by the keys ‘0.0’, ‘0.1’, ‘1.0’, ‘1.1’.





Store interface

- Any storage system can be used with Zarr if it can provide a key/value interface.
  - Keys are strings, values are bytes.
- In Python, we use the MutableMapping interface.
  - `__getitem__`
  - `__setitem__`
  - `__iter__`
- I.e., anything dict-like can be used as a Zarr store.

E.g., ZipStore implementation

```
class ZipStore(MutableMapping):

    def __init__(self, path, ...):
        self.zf = zipfile.ZipFile(path, ...)

    def __getitem__(self, key):
        with self.zf.open(key) as f:
            return f.read()

    def __setitem__(self, key, value):
        self.zf.writestr(key, value)

    def __iter__(self):
        for key in self.zf.namelist():
            yield key
```

(Actual implementation is slightly more complicated, but this is the essence.)

Parallel computing with Zarr

- A Zarr array can have multiple concurrent readers*.
- A Zarr array can have multiple concurrent writers*.
- Both multi-thread and multi-process parallelism are supported.
- GIL is released during critical sections (compression and decompression).

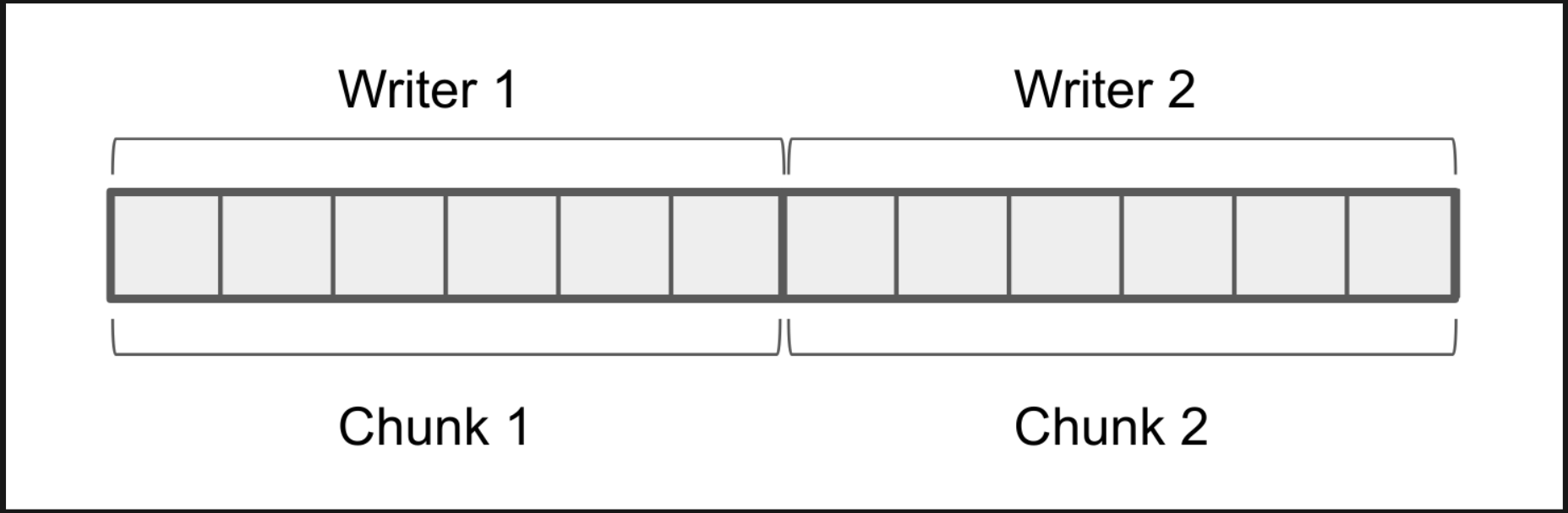
* Depending on the store.

Dask + Zarr

```
import dask.array as da
import zarr

# set up input
store = ...  # some Zarr store
root = zarr.group(store)
big = root['big']
big = da.from_array(big)

# define computation
output = big * 42 + ...

# if output is small, compute to memory
o = output.compute()

# if output is big, compute and write directly to Zarr
da.to_zarr(output, store, component='output')
```

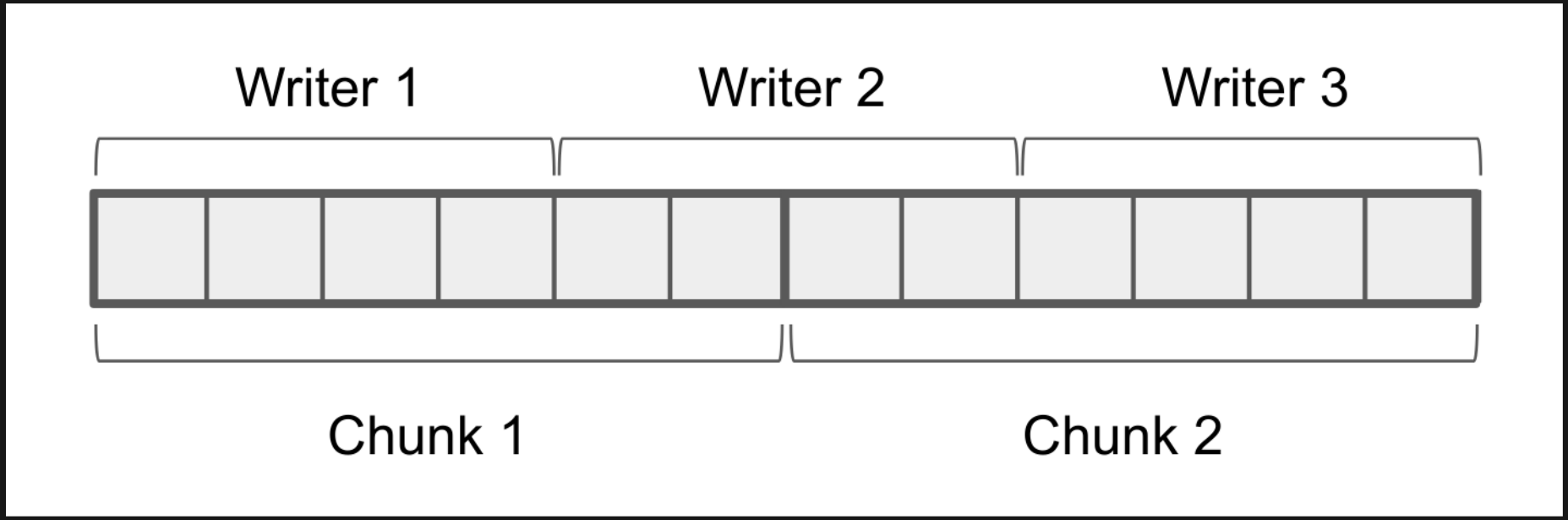
See docs for `da.from_array()`, `da.from_zarr()`, `da.to_zarr()`, `da.store()`.

Write locks?



- If each writer is writing to a different region of an array, and all writes are **aligned with chunk boundaries**, then locking is **not required**.

Write locks?



- If each writer is writing to a different region of an array, and writes are **not aligned** with chunk boundaries, then locking **is required** to avoid contention and/or data loss.

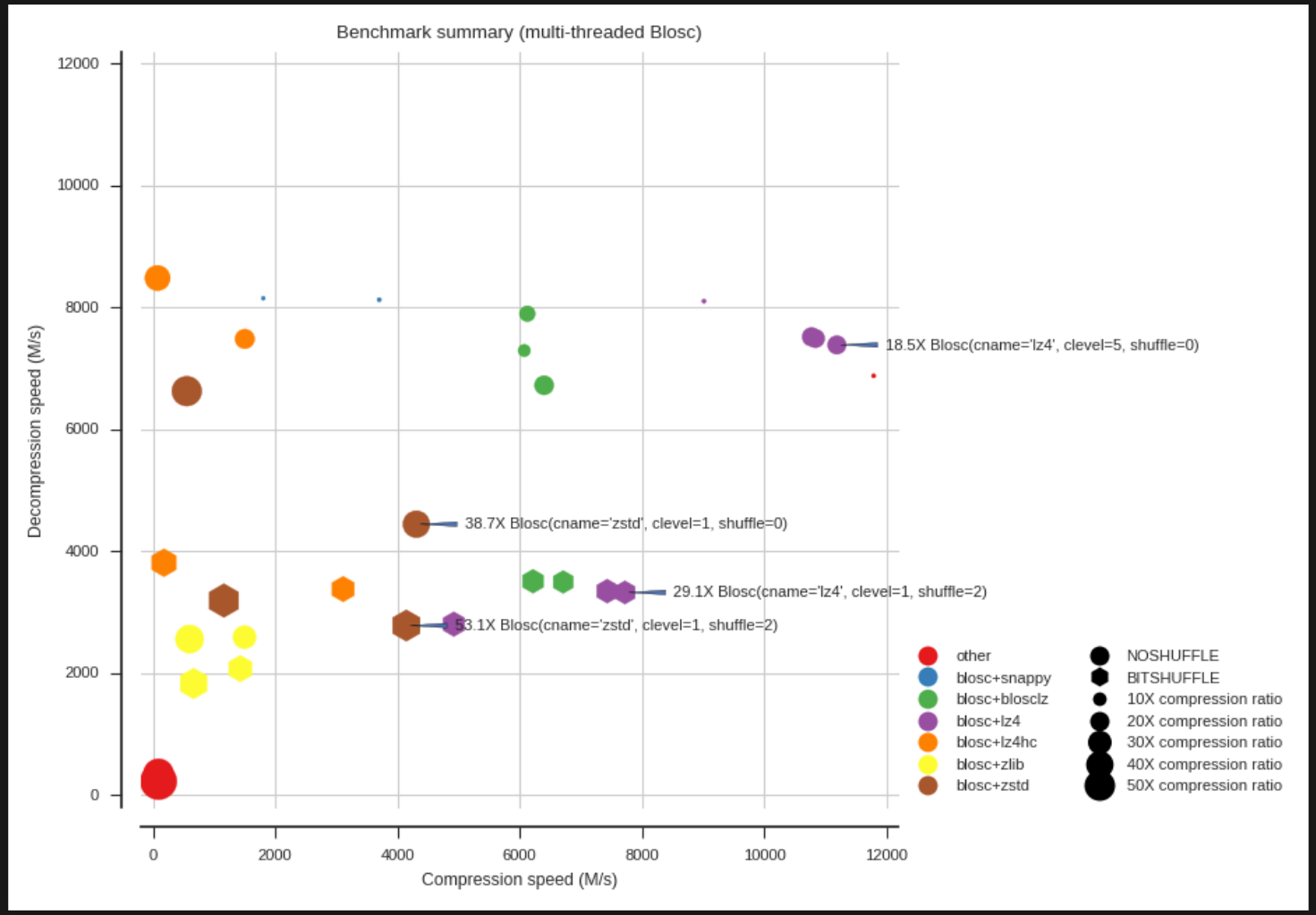
Write locks?

- Zarr does support chunk-level write locks for either multi-thread or multi-process writes.
- But generally easier and better to align writes with chunk boundaries where possible.

See Zarr tutorial for further info on synchronisation.

Pluggable compressors

Compressor benchmark (genomic data)



http://alimanfoo.github.io/2016/09/21/genotype-compression-benchmark.html

Available compressors (via numcodecs)

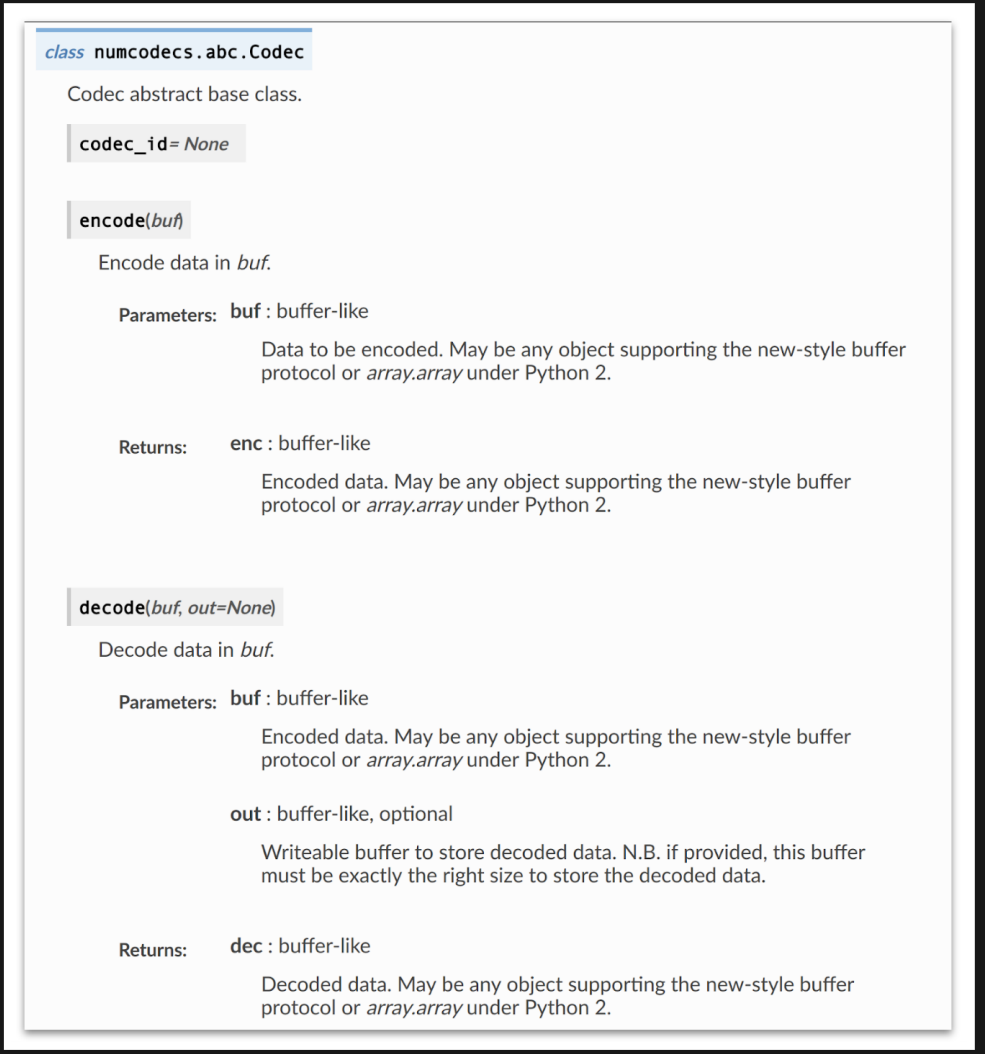
Blosc, Zstandard, LZ4, Zlib, BZ2, LZMA, ...

```
import zarr
from numcodecs import Blosc

store = zarr.DirectoryStore('example.zarr')
root = zarr.group(store)
compressor = Blosc(cname='zstd', clevel=1, shuffle=Blosc.BITSHUFFLE)
big2 = root.zeros('big2',
                  shape=(100_000_000, 100_000_000),
                  chunks=(10_000, 10_000),
                  dtype='i4',
                  compressor=compressor)
```

Compressor interface

The numcodecs Codec API defines the interface for filters and compressors for use with Zarr.

Built around the Python buffer protocol.



class **numcodecs.abc.Codec**

Codec abstract base class.

codec_id= *None*

encode(*buf*)

Encode data in *buf*.

Parameters: **buf** : buffer-like

Data to be encoded. May be any object supporting the new-style buffer protocol or *array.array* under Python 2.

Returns: **enc** : buffer-like

Encoded data. May be any object supporting the new-style buffer protocol or *array.array* under Python 2.

decode(*buf*, *out=None*)

Decode data in *buf*.

Parameters: **buf** : buffer-like

Encoded data. May be any object supporting the new-style buffer protocol or *array.array* under Python 2.

out : buffer-like, optional

Writeable buffer to store decoded data. N.B. if provided, this buffer must be exactly the right size to store the decoded data.

Returns: **dec** : buffer-like

Decoded data. May be any object supporting the new-style buffer protocol or *array.array* under Python 2.

```
class Zlib(Codec):

    def __init__(self, level=1):
        self.level = level

    def encode(self, buf):

        # normalise inputs
        buf = ensure_contiguous_ndarray(buf)

        # do compression
        return zlib.compress(buf, self.level)

    def decode(self, buf, out=None):

        # normalise inputs
        buf = ensure_contiguous_ndarray(buf)
        if out is not None:
            out = ensure_contiguous_ndarray(out)

        # do decompression
        dec = zlib.decompress(buf)

        return ndarray_copy(dec, out)
```

Zarr specification

Zarr storage specification version 2

This document provides a technical specification of the protocol and format used for storing Zarr arrays. The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119.

Status

This specification is the latest version. See Specifications for previous versions.

Storage

A Zarr array can be stored in any storage system that provides a key/value interface, where a key is an ASCII string and a value is an arbitrary sequence of bytes, and the supported operations are read (get the sequence of bytes associated with a given key), write (set the sequence of bytes associated with a given key) and delete (remove a key/value pair).

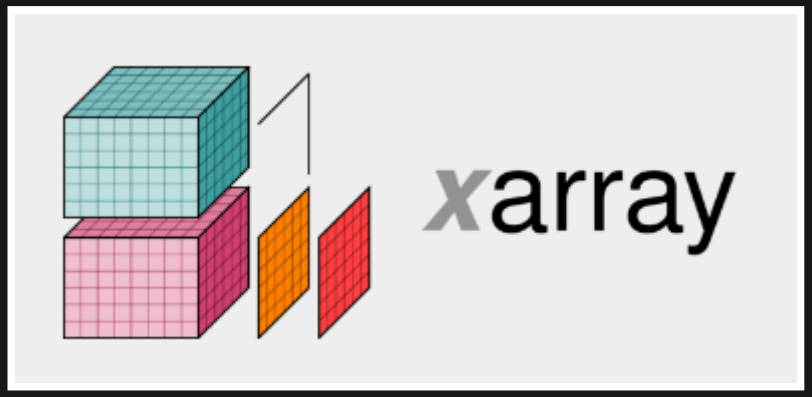
For example, a directory in a file system can provide this interface, where keys are file names, values are file contents, and files can be read, written or deleted via the operating system. Equally, an S3 bucket can provide this interface, where keys are resource names, values are resource contents, and resources can be read, written or deleted via HTTP.

Other Zarr implementations

- z5 - C++ implementation using xtensor
- Zarr.jl - native Julia implementation
- ndarray.scala - Scala implementation
- WIP: NetCDF and native cloud storage access via Zarr

Integrations and applications

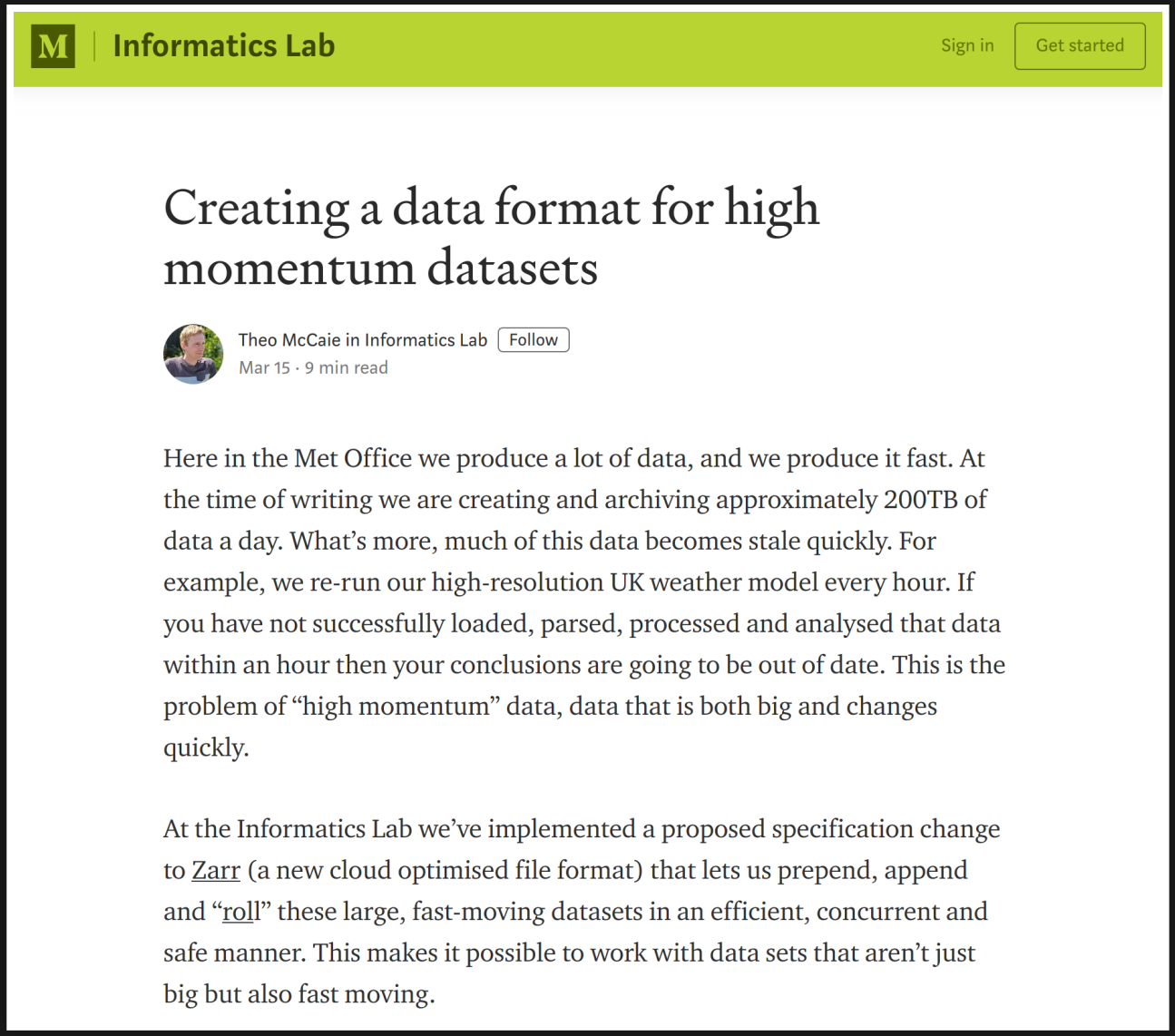
Xarray, Intake, Pangeo



- xarray.open_zarr(), xarray.Dataset.to_zarr().
- Intake project for data catalogs has intake-xarray plugin with Zarr support.
- Used by Pangeo for their cloud datastore ...

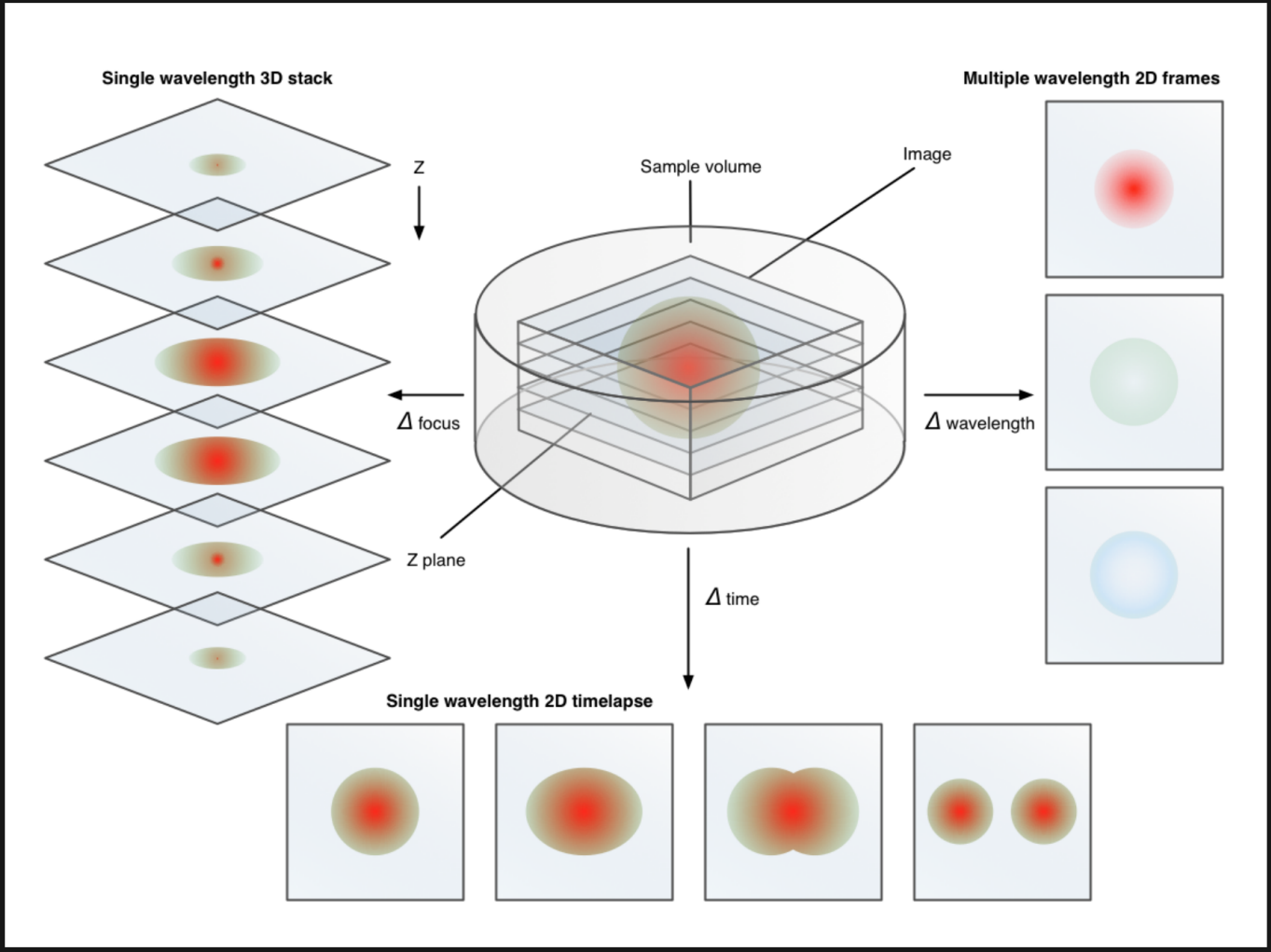
```
import intake
cat_url = 'https://raw.githubusercontent.com/pangeo-data/pangeo-data
cat = intake.Catalog(cat_url)
ds = cat.atmosphere.gmet_v1.to_dask()
```

(Here's the underlying data catalog entry.)



https://medium.com/informatics-lab/creating-a-data-format-for-high-momentum-datasets-a394fa48b671

Microscopy (OME)



See OME's position regarding file formats.

Single cell biology

- Work by Laserson lab using Zarr with ScanPy and AnnData to scale single cell gene expression analyses.
- The Human Cell Atlas data portal uses Zarr for storage of gene expression matrices.
- Use Zarr for image-based transcriptomics (starfish)?

Future

- Zarr/N5 convergence.
- Zarr protocol spec v3.
- Community!

Credits

- Zarr core development team.
- Everyone who has contributed code or raised or commented on an issue or PR, thank you!
- UK MRC and Wellcome Trust for supporting @alimanfoo.
- Zarr is a community-maintained open source project - please think of it as yours!