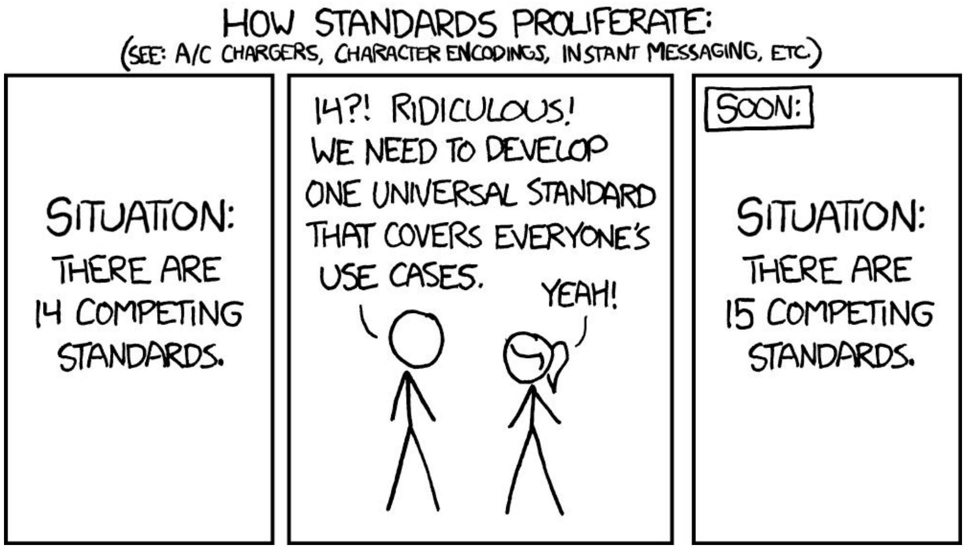
Using a HDF5 file as Zarr v3 Shard

Mark Kittisopikul, Ph.D.
Software Engineer III
Scientific Computing Software
Janelia Research Campus
Howard Hughes Medical Institute

HDF5 User Group, May 26, 2025

Why combine multiple file formats?

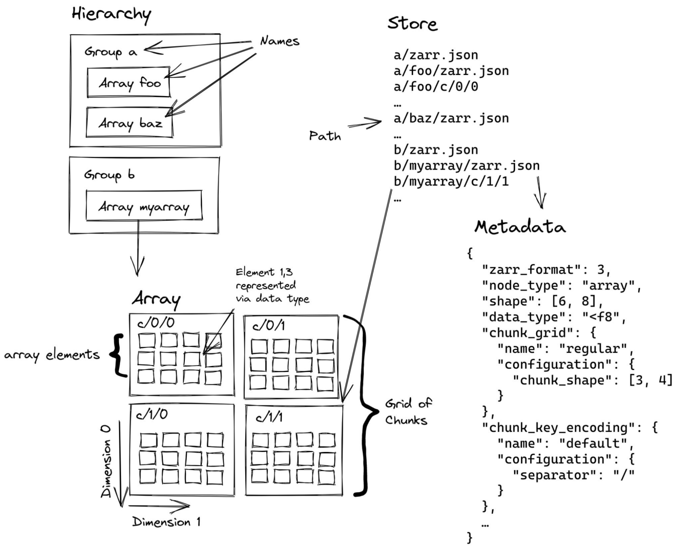
- Avoid data duplication
  - Large datasets could be terabytes, petabytes, or exabytes in scale
  - We cannot afford to have multiple copies
- Make it easier for users to read using their favorite APIs
  - Users may have restricted access to one API but still need to access the same data



https://xkcd.com/927

Zarr v3 is a cloud optimized chunk-based hierarchical array storage specification

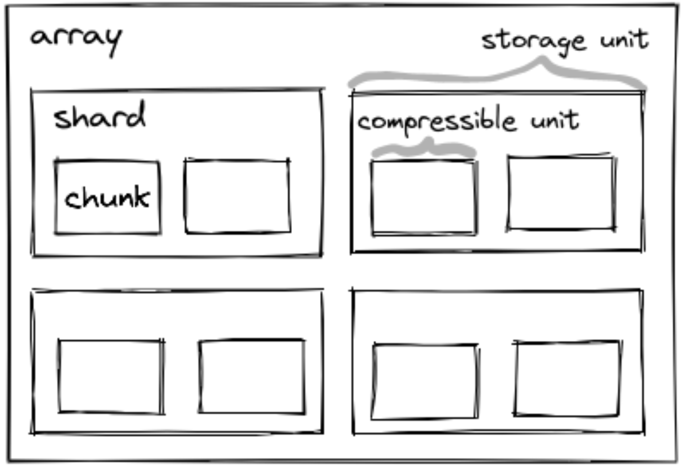
- Metadata are stored as JSON **zarr.json** files for each group and array
- Chunks are stored as individual keys (files) in key-value store (a filesystem)
    - On a file system, chunks are individual files.
- For small chunks (e.g. 32x32x32) in a large array (e.g. 4096x4096x4096), this could result in millions of files
    - Efficient access requires optimized file systems



https://zarr-specs.readthedocs.io/en/latest/v3/core/index.html#

A Zarr v3 shard is a codec to subdivide a single chunk into smaller inner chunks

- Many chunks can exist in a single file.
- Therefore, we can reduce the number of files required. Example:
  - Array size: 4096 x 4096 x 4096
  - Shard size: 1024 x 1024 x 1024
  - Chunk size: 32 x 32 x 32
- Inner chunks can be individual compressed



https://zarr-specs.readthedocs.io/en/latest/v3/codecs/sharding-indexed/index.html

A Zarr v3 shard chunk index exists at either the beginning or end of the shard

- The size of the chunk index can be calculated directly from information in the zarr.json file
  - nChunks x 16 bytes + 4 bytes
  - The 4-byte checksum of the chunk index is calculated using CRC32c
  - Retrievable using a single HTTP GET request with a byte-range header.

CRC32c

```
| chunk (0, 0)    | chunk (0, 1)    | chunk (1, 0)    | chunk (1, 1)    |          |
| offset | nbytes | offset | nbytes | offset | nbytes | offset | nbytes | checksum |
| uint64 | uint64 | uint64 | uint64 | uint64 | uint64 | uint64 | uint64 | uint32   |
```

The Zarr v3 shard index is similar to a HDF5 Fixed Array Data Block, differ by 14 or 18 bytes

Fields: Fixed Array Data Block

Field Name	Description
Signature	The ASCII character string " `FADB` " is used to indicate the beginning of a Fixed Array data block. This gives file consistency checking utilities a better chance of reconstructing a damaged file.
Version	This document describes version 0.
Client ID	The ID for identifying the client of the Fixed Array: ID / Description `0` Non-filtered dataset chunks `1` Filtered dataset chunks `2+` Reserved.
Header Address	The address of the Fixed Array header. Principally used for file integrity checking.
Page Bitmap	A bitmap indicating which data block pages are initialized. Exists only if the data block is paged.
Elements	Contains the elements stored in the data block and exists only if the data block is not paged. There are two element types: ID / Description `0` Non-filtered dataset chunks `1` Filtered dataset chunks
Checksum	The checksum for the Fixed Array data block.

14 bytes

Jenkins

HDF5

Layout: Data Block Element for Filtered Dataset Chunk

byte	byte	byte	byte	
Address^{O}				64-bits
Chunk Size *(variable size; at most 8 bytes)*				32-bits
Filter Mask				32-bits

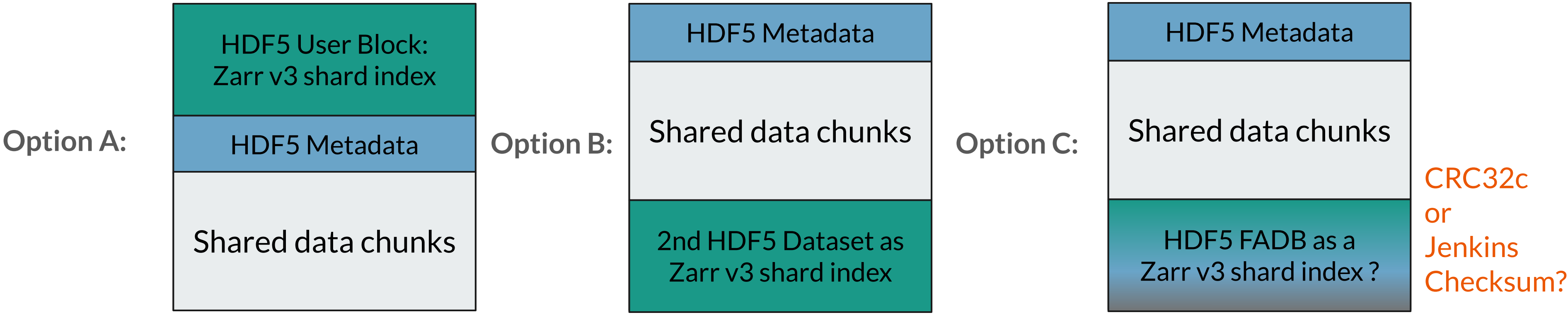
Zarr

```
| chunk (0, 0)     |
| offset | nbytes  |
| uint64 | uint64  |
```

```
| CRC32c   |
| checksum |
| uint32   |
```

Formatting a HDF5 File as a Zarr v3 shard?

- We need to place the Zarr v3 shard index at the beginning or end of the file
- Options
  - A: Put the Zarr v3 shard index in the HDF5 User Block at the beginning of the file
  - B: Put the Zarr v3 shard index into a dataset at the end of the file
  - C: Relocate the HDF5 Fixed Array Data Block to the end of the file



Zarr v3 sharding is similar a HDF5 Virtual Dataset

- Virtual datasets are a HDF5 feature that allows part of a dataset to exist as a dataset in another file (~Zarr v3 shard)
- A Zarr v3 shard is analogous to a file with a single chunked source dataset

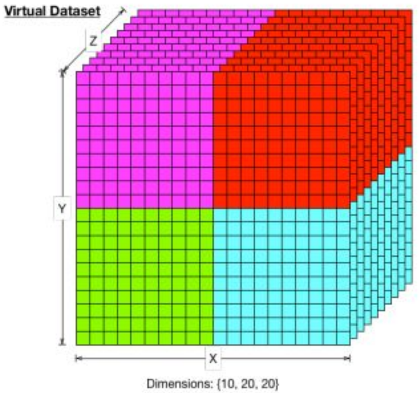
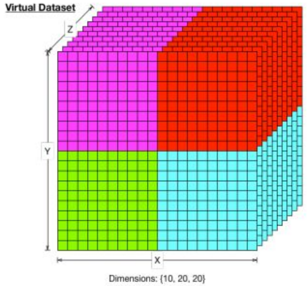


Figure 1: Mapping Source Datasets to Virtual Dataset

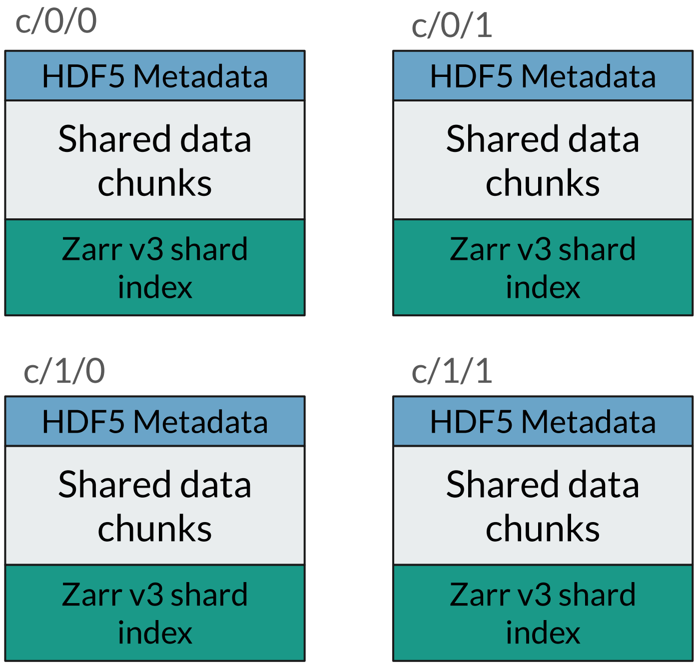
Combined Zarr Array as a HDF5 Virtual Dataset

c/0/0 c/0/1



c/1/0 c/1/1

Figure 1: Mapping Source Datasets to Virtual Dataset

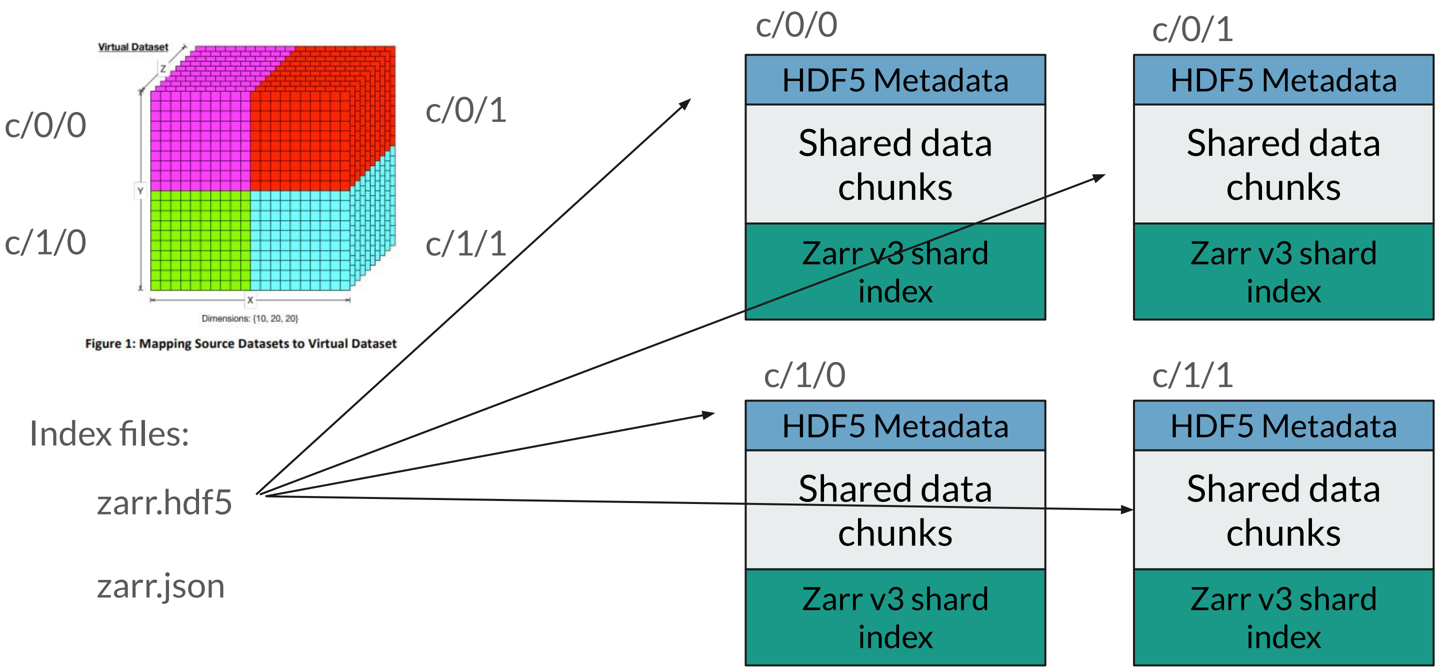


Index files:

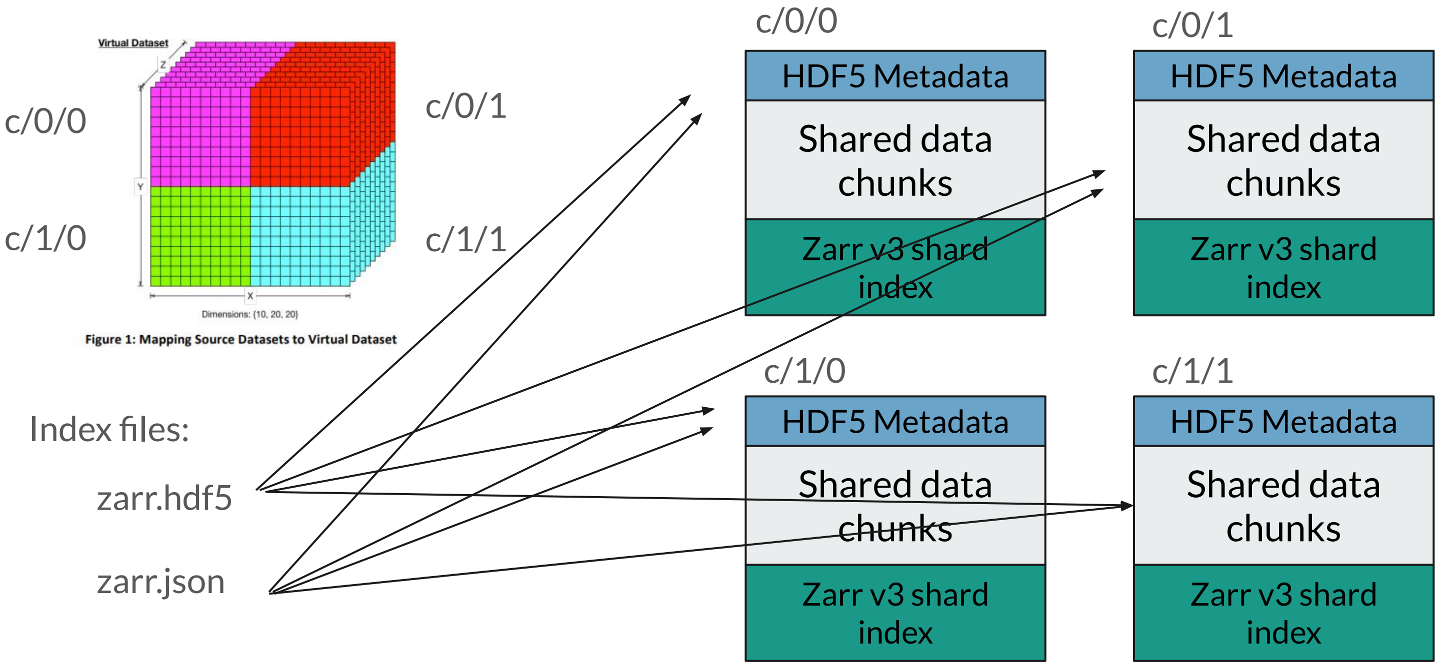
zarr.hdf5

zarr.json

Combined Zarr Array as a HDF5 Virtual Dataset



Combined Zarr Array as a HDF5 Virtual Dataset

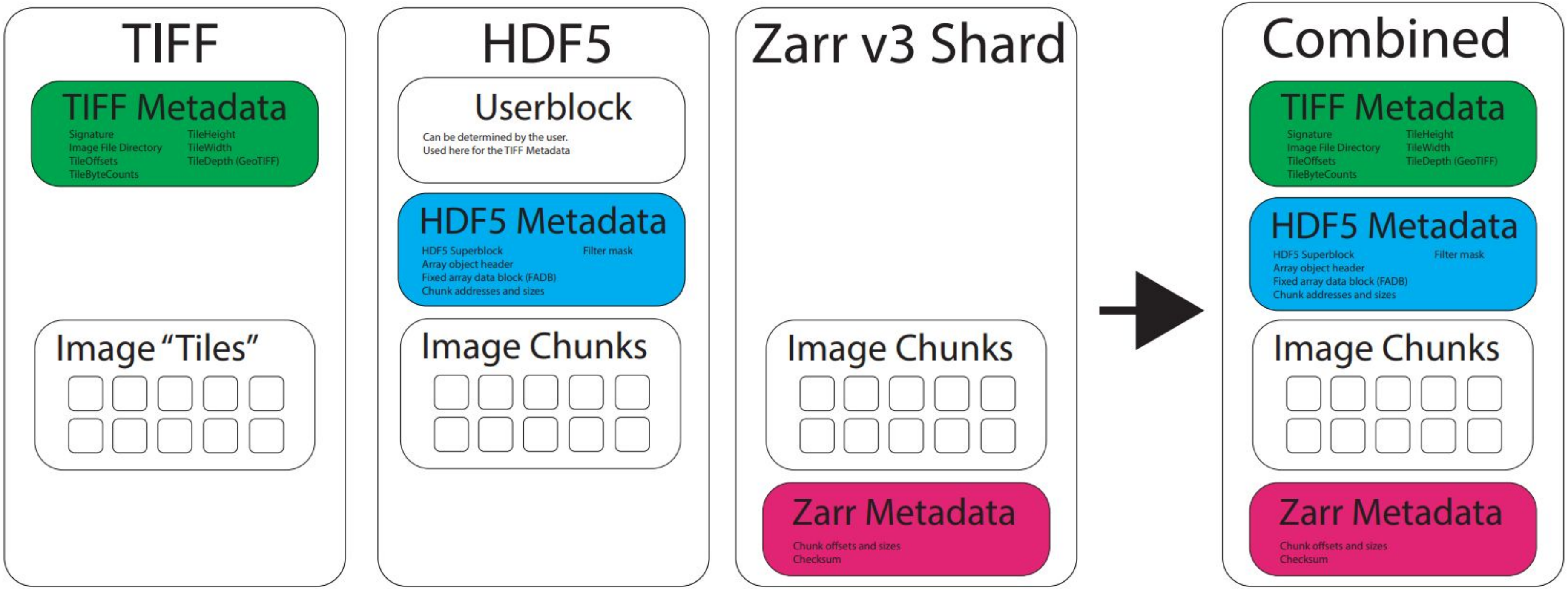


Summary

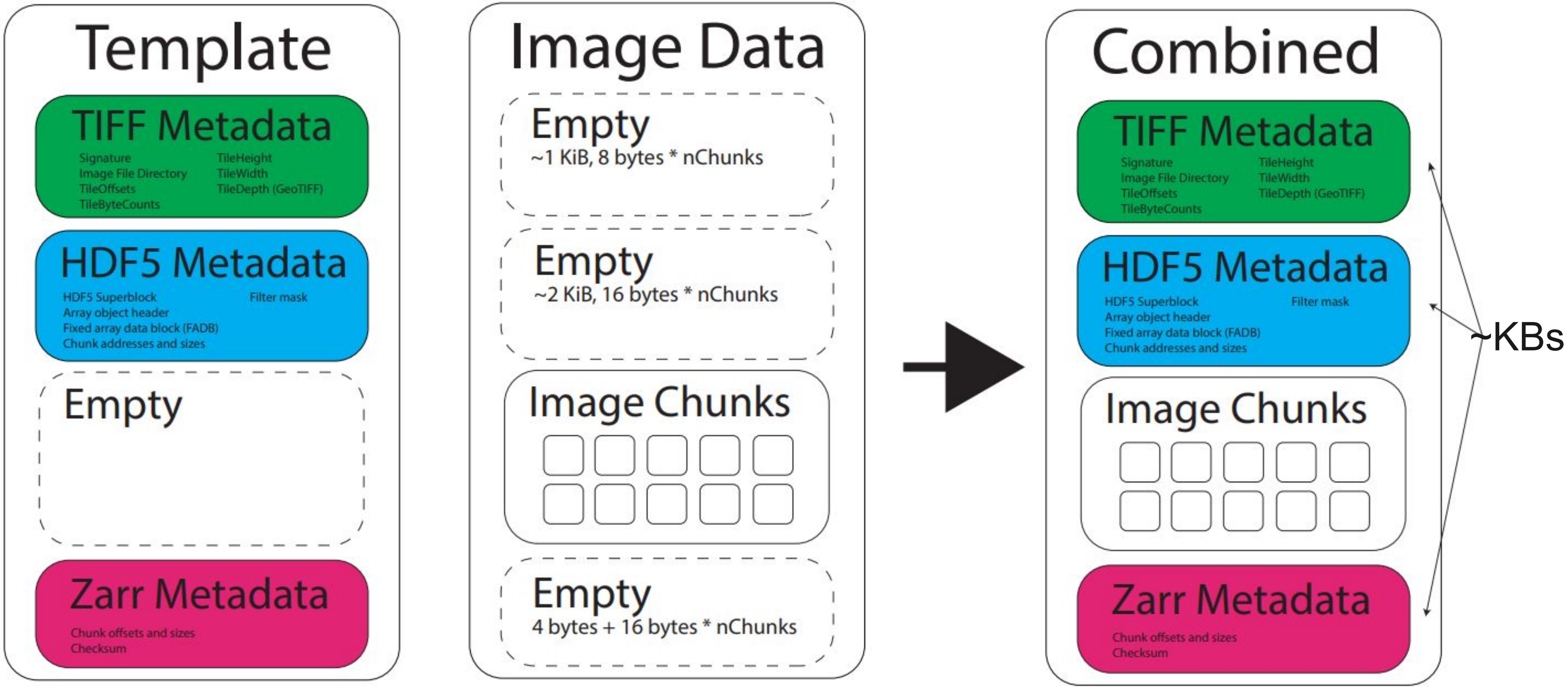
- Zarr v3 shards partition chunks into small inner chunks.
- The resulting arrangement is similar to a HDF5 Virtual Dataset.
- A “file” could be both a valid HDF5 file and a Zarr v3 shard
  - The Zarr v3 shard index could exist in a HDF5 file either as
    - A user block at the beginning of the file OR
    - An extra contiguous dataset at the end of the file
  - A merged FADB and Zarr v3 shard index would require alignment of 32-bit checksums
    - HDF5 adopts CRC32c as a checksum
    - Zarr adopts Jenkin’s lookup3 as a codec
- Alternative: A Zarr v3 virtual file driver for HDF5?
- Bonus (time permitting): Combining TIFF, HDF5, and Zarr v3
  - Jupyter notebook demonstration

Bonus topic: Combining TIFF, HDF5, and Zarr v3

Could we combine TIFF, HDF5, and Zarr?



Can microscopists implement simply and efficiently?



Demo: Reading the same file via distinct packages

I've created a single file "demo.hdf5.zarr.tiff" that can be read by distinct Python packages:

- h5py (HDF5)
- libtiff (TIFF)
- tensorstore (Zarr v3)

https://github.com/mkitti/simple_image_formats

```
In [6]: import h5py
        with h5py.File("demo/demo.hdf5.zarr.tiff") as h5f:
            h5data = h5f["data"][:]

        h5data

Out[6]: array([[    0,     0,     0, ..., 16382, 16382, 16382],
               [    0,     0,     0, ..., 16382, 16382, 16382],
               [    0,     0,     0, ..., 16382, 16382, 16382],
               ...,
               [32766, 32766, 32766, ..., 49150, 49150, 49150],
               [32766, 32766, 32766, ..., 49150, 49150, 49150],
               [32766, 32766, 32766, ..., 49150, 49150, 49150]], dtype=uint16)

In [7]: from libtiff import TIFF
        tif = TIFF.open("demo/demo.hdf5.zarr.tiff", "r")
        tiff_data = tif.read_image()
        tif.close()
        tiff_data

Out[7]: array([[    0,     0,     0, ..., 16382, 16382, 16382],
               [    0,     0,     0, ..., 16382, 16382, 16382],
               [    0,     0,     0, ..., 16382, 16382, 16382],
               ...,
               [32766, 32766, 32766, ..., 49150, 49150, 49150],
               [32766, 32766, 32766, ..., 49150, 49150, 49150],
               [32766, 32766, 32766, ..., 49150, 49150, 49150]], dtype=uint16)

In [8]: import tensorstore as ts
        ts.open({
            "driver": "zarr3",
            "kvstore": {
               "driver": "file",
               "path": "demo/test.zarr/"
            },
        }).result().read().result()

Out[8]: array([[    0,     0,     0, ..., 16382, 16382, 16382],
               [    0,     0,     0, ..., 16382, 16382, 16382],
               [    0,     0,     0, ..., 16382, 16382, 16382],
               ...,
               [32766, 32766, 32766, ..., 49150, 49150, 49150],
               [32766, 32766, 32766, ..., 49150, 49150, 49150],
               [32766, 32766, 32766, ..., 49150, 49150, 49150]], dtype=uint16)
```

We can modify the data with h5py (via HDF5) ...

In [9]:

```
import h5py
import shutil
with h5py.File("demo/demo.hdf5.zarr.tiff", "r+") as h5f:
    h5f["data"][:128,:128] = 1
    h5f["data"][:128,128:] = 2
    h5f["data"][128:,:128] = 3
    h5f["data"][128:,128:] = 4

# copy to the zarr shard, consider a symlink on Linux systems
shutil.copyfile("demo/demo.hdf5.zarr.tiff", "demo/test.zarr/c/0/0")
```

Out[9]: 'demo/test.zarr/c/0/0'

… Then read the modified data from all three libraries as a TIFF, HDF5 file, or a Zarr v3 shards

We can make the latest standards cooperate rather than compete.

Is this the best approach? Should we address this via file systems (FUSE), APIs (N5), or services?

In [10]:

```python
with h5py.File("demo/demo.hdf5.zarr.tiff") as h5f:
    print(h5f["data"][:])
```

```
[[1 1 1 ... 2 2 2]
 [1 1 1 ... 2 2 2]
 [1 1 1 ... 2 2 2]
 ...
 [3 3 3 ... 4 4 4]
 [3 3 3 ... 4 4 4]
 [3 3 3 ... 4 4 4]]
```

In [11]:

```python
tif = TIFF.open("demo/demo.hdf5.zarr.tiff", "r")
print(tif.read_image())
tif.close()
```

```
[[1 1 1 ... 2 2 2]
 [1 1 1 ... 2 2 2]
 [1 1 1 ... 2 2 2]
 ...
 [3 3 3 ... 4 4 4]
 [3 3 3 ... 4 4 4]
 [3 3 3 ... 4 4 4]]
```

In [12]:

```python
ts.open({
    "driver": "zarr3",
    "kvstore": {
       "driver": "file",
       "path": "demo/test.zarr/"
    },
}).result().read().result()
```

Out[12]:

```
array([[1, 1, 1, ..., 2, 2, 2],
       [1, 1, 1, ..., 2, 2, 2],
       [1, 1, 1, ..., 2, 2, 2],
       ...,
       [3, 3, 3, ..., 4, 4, 4],
       [3, 3, 3, ..., 4, 4, 4],
       [3, 3, 3, ..., 4, 4, 4]], dtype=uint16)
```

Implementation details...

- HDF5 metadata can be consolidated by using a large enough meta_block_size
- HDF5 chunk information (offset and nbytes) can be extracted efficiently using H5chunk_iter