Zarr storage specification version 2:
Cloud-optimized persistence using Zarr

Status of this Memo

This Request for Comment (RFC) provides a description of a proposed Standard/Convention for use in NASA Earth science data systems.

Change Explanation

This document is not a revision to an earlier version.

Copyright Notice

Copyright © 2023 United States Government as represented by the Administrator of the National Aeronautics and Space Administration. No copyright is claimed in the United States under Title 17, U.S. Code. All Other Rights Reserved.

Abstract

This document designates the Zarr storage specification version 2 as a cloud-optimized data standard for NASA Earth science data systems.

Table of Contents

1 INTRODUCTION **2**
1.1 MOTIVATION 2
1.2 Evidence of Implementation 3
2 ZARR STRUCTURE **3**
2.1 METADATA 5
2.2 Data 6
3 API 6
3.1 Implementations 6
3.2 RELEVANT LIBRARIES AND EXTENSIONS 9
4 INTEROPERABILITY AND APPLICABILITY CONSIDERATIONS **9**
5 FUTURE VERSIONS OF THE SPECIFICATION **10**
6 REFERENCES **10**
7 AUTHORS' ADDRESSES **11**

1 Introduction

Zarr is a community project to develop specifications and software for storage of large N-dimensional typed arrays. A particular focus of Zarr is to provide support for storage using distributed systems like cloud object stores, and to enable efficient I/O for parallel computing applications [1].

1.1 Motivation

One of EOSDIS' primary motivations for migrating our data holdings to the cloud is to enable data-adjacent computing capabilities at scale for all. By data-adjacent we mean that the data is close, network-wise, to the compute you bring to bear on that data.
With the rapid explosion of data volumes, and the corresponding explosion in the compute necessary to produce science from that data, it is of paramount importance that users in the cloud can access cloud-hosted data in an efficient and performant manner. Current EOSDIS archival data formats are optimized for storage rather than access. Consequently, tooling around those datasets is optimized for traditional file systems on local storage.
Given the different latency profiles of cloud and local storage, a cloud-optimized format should strive to minimize the number of read operations whilst also minimizing the volume of data accessed.
Since EOSDIS' traditional formats, such as HDF4, HDF-EOS, HDF5, NetCDF-3, and NetCDF-classic, are optimized for storage they produce large files. Quite often, a user will only need a small percentage of the file to do their analysis. In order to determine which part of the file they need, in the case of NetCDF and non-optimized HDF, they need to read a significant percentage of the file. When accessing the file through a distributed network rather than a local file system this pattern is the opposite of the optimal case of minimizing reads and volume of data accessed.
A cloud-optimized format needs to support the following,

- High-throughput distributed I/O
- Parallel processing at the process and thread level
- Chunking - the ability to split data into manageable, addressable parts along specific dimensions. This allows the user or client to make performant decisions on what data they access within a data store.
- Chunk compression - the ability to compress individual chunks rather than the dataset as a whole so that decompression of data does not require access to the entire dataset.

One such format, optimized for object stores such as Amazon AWS S3 [3], is Zarr. Zarr is specifically useful for the earth science domain as it supports large, multi-dimensional data arrays and is compatible with a variety of existing data science tooling such as DASK [4] and XArray [5].
The Zarr data format utilizes the following features to achieve cloud-optimized data access,

- Random access with comparatively few communications
- Consolidated metadata in a common plain-text format (JSON). Other existing archival formats tend to place their metadata throughout the file space in binary formats.
- Chunking across any dimension
- 1:1 file/chunk ratio to ease parallelism. Other existing archival formats use single binary files and tend to provide library support that is thread-safe rather than thread-optimized.

- Chunk-based compression

1.2 Evidence of Implementation

1.2.1 Pangeo-forge

The goal of the Pangeo Forge project [6] is to make it easy to extract data from traditional data repositories and deposit them in cloud object storage in ARCO formats. It uses the Zarr format to achieve that goal. It has, as of writing, produced 42 datasets in the Zarr format on AWS S3 using various earth science sources such as NASA, NOAA, DOE, The Australian Bureau of Meteorology and the UK Met Office. It is an open source, community-driven effort that has integrations with NASA EOSDIS' CMR for finding canonical NASA earth science datasets as inputs for Zarr stores.
It provides recipes for generating Zarr stores from archival data stores in a variety of formats (such as NetCDF and HDF).

1.2.2 Giovanni in the cloud

The Geospatial Interactive Online Visualization And Analysis Infrastructure (Giovanni), developed at NASA's GES DISC, is using Zarr to provide time and area averaging services for GES DISC data. They achieve this in a performant manner by using a single grid-cell time series service based on the Zarr storage format in AWS S3. GES DISC currently holds 1.82TB of data in Zarr format, representing 6 variables with a projected 20TB of data representing 2000 variables in the GES DISC archive.

1.2.3 AWS Open Data Registry

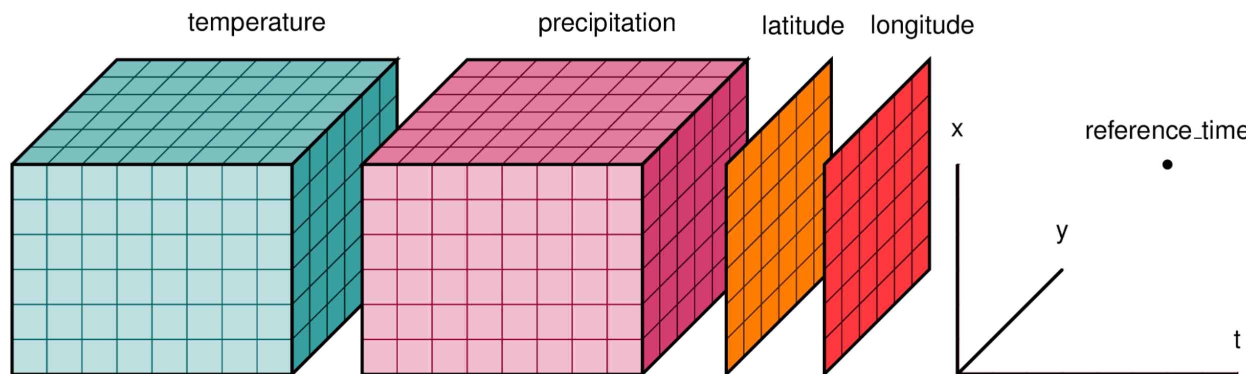
There are a variety of Zarr stores related to Earth Science data in AWS' Open Data Registry [7]. One, oft-cited, example is the Multi-Scale Ultra High Resolution (MUR) Sea Surface Temperature (SST) dataset [8] which provides a Zarr store for PO.DAAC's global, gap-free, gridded, daily 1 km Sea Surface Temperature dataset.

1.2.4 Zarr stores and dynamic imagery generation

The access performance of Zarr stores opens up the possibility of imagery services providing functionality without persisting intermediate assets. Historically, imagery services such as NASA GIBS [9] have generated persistent stores of static imagery from archival data to provide WMS and WMTS services. Zarr storage in the cloud allows the possibility of generating the imagery required for their services on-demand from a Zarr store.
This architectural change is also being pursued by NASA IMPACT's VEDA [10] to publish and visualize NASA's Earthdata Zarr archives and deliver interoperable APIs for its data stores to support dynamic data visualization and storytelling.

2 Zarr Structure

Like HDF5, the Zarr format is designed to store an arbitrary number of array variables of arbitrary dimensionality in a chunked, binary format, along with arbitrary metadata. Unlike HDF5, a Zarr store is not a single file but a nested directory structure: Each variable is contained within a single directory that contains the variable's metadata files, in plain-text JSON, and a set of compressed files representing the chunks of the variable's data.
For example, measuring temperature and precipitation over a spatial and temporal area could be visualized as follows,



[1]

Each block in the first array would represent a temperature at a given spatial location (expressed in latitude and longitude) for a given time. The second block would represent precipitation in the same manner.
The Zarr format consists of metadata in the json format that enables simple and efficient random access to a 'Zarr store' which contains a set of compressed, binary data objects arranged in a manner that matches the chunking scheme described in the metadata. This allows each chunk to be accessed, decompressed and analyzed in isolation.

[1] https://xarray.dev/

2.1 Metadata

Information about the structure and composition of the arrays in a datastore is represented in a number of json files in the root and groups of the store.

2.1.1 .zarray

This file describes the composition of an array including the dimensions or shape, the chunking strategy and the compression used on each chunk.

```
{'shape': [8, 6, 6],
 'chunks': [4, 3],
 'compressor': {'blocksize': 0,
                'clevel': 5,
                'cname': 'lz4',
                'id': 'blosc',
                'shuffle': 1},
 'dtype': '<f8',
 'fill_value': 0.0,
 'filters': None,
 'order': 'C',
 'zarr_format': 2}
```

2.1.2 .zattrs

This file describes attributes associated with an array. For example, the CF standard name of the measurement described and its units.

```
{'standard_name': 'air_temperature',
'units': 'degC'}
```

2.1.3 .zmetadata

In order to optimize performance for network distributed stores containing multiple datasets/groups, Zarr allows you to consolidate metadata associated with multiple arrays (.zarray, .zattrs) in a single top level artifact (.zmetadata). This means that the entire structure of the Zarr store can be determined with a single request.

2.2 Data

Each array contains one or more chunks. Each chunk is a binary, compressed object/file. That chunk will contain a segment of the array defined by the .zarray file. These chunks are named according to their location in the array.

3 API

Zarr provides library support in numerous languages to read and write Zarr to a number of persistence stores [11]. Examples of API functionality are shown in Python for simplicity. It should be noted that file access is generally achieved by other libraries that use Zarr as a data store. Those libraries are detailed in the ‘relevant libraries and extensions’ section of this document.

3.1 Implementations

- Python: https://github.com/zarr-developers/zarr-python
- Julia: https://github.com/JuliaIO/Zarr.jl

- C++: https://github.com/constantinpape/z5
- Scala: https://github.com/lasersonlab/ndarray.scala
- Java: https://github.com/bcdev/jzarr

3.1.1 Creating and chunking a Zarr store
The following is a trivial example of a Zarr store creation to demonstrate that chunking characteristics can be assigned on creation.

```
>import zarr
>z=zarr.zeros((10000,10000),chunks=(1000,1000), dtype='i4')
>z
<zarr.core.Array (10000, 10000) int32>
```

3.1.2 Compression
Each chunk in a Zarr store can be compressed. This means that data users can take advantage of the lower latency compression affords but still limit the amount of data they access. Data curators can choose from a variety of compression algorithms to suit their needs.

```
>from numcodecs import Blosc
>compressor=Blosc(cname='zstd',clevel=3,shuffle=Blosc.BITSH
UFFLE)
>data=np.arange(100000000,dtype='i4').reshape(10000,10000)
>z=zarr.array(data,chunks=(1000,1000),compressor=compressor
)
>z.compressor
Blosc(cname='zstd',clevel=3,shuffle=BITSHUFFLE,blocksize=0)
```

3.1.3 Reading from and writing to a Zarr store
In python, writing to and reading from a Zarr store can be achieved in the same way one would do it using NumPy.

```
>z[0:0] = 42
>z[0, 0]
42
>z[0,1]
0
```

3.1.4 Storage options
Python Zarr can use any object that implements the MutableMapping interface from the collections module in the Python standard library as the store for a group or an array.
3.1.4.1 Local storage
The simplest local storage is a file system. Zarr refers to this as a directory store.

```
> store = zarr.DirectoryStore('data/example.zarr')
```

But other local stores can be used, such as Zip Files or relational databases.

```
> store = zarr.ZipStore('data/example.zip', mode='r')
> store = zarr.SQLiteStore('data/example.sqldb')
```

3.1.4.2 Distributed storage

Zarr provides interfaces to different distributed storage mechanisms, a key factor in our recommendation.

```
>import s3fs
>import zarr
>s3=s3fs.S3FileSystem(anon=True,client_kwargs=dict(region_name
='eu-west-2'))
>store=s3fs.S3Map(root='zarr-demo/store', s3=s3, check=False)
>root = zarr.group(store=store)
>z = root['foo/bar/baz']

>import azure.storage.blob
>container_client = azure.storage.blob.ContainerClient(...)
>store=zarr.ABSStore(client=container_client,prefix='zarr-
testing')
>root=zarr.group(store=store, overwrite=True)
```

3.1.5 Parallel computing

Zarr arrays have been designed for both multiple concurrent read and multiple concurrent write operations in parallel computations. Multiple concurrent write operations may occur, if each writer is updating a different chunk.
Both multi-threaded and multi-process parallelism are possible. If each worker in a parallel computation is writing to a separate chunk, then no synchronization is required.
If concurrent writes are required across chunks then Zarr can be configured to support that but performance will suffer as a consequence.

```
>z=zarr.zeros((10000,10000),chunks=(1000,1000), dtype='i4',
synchronizer=zarr.ThreadSynchronizer())
>synchronizer=zarr.ProcessSynchronizer('data/example.sync')
>z=zarr.open_array('data/example',mode='w',shape=(10000,
10000),chunks=(1000,1000), dtype='i4',synchronizer=synchronizer)
```

3.2 Relevant libraries and extensions

The relative simplicity of the Zarr format dramatically facilitates the implementations of Zarr drivers in different programming languages. All that is required to implement a Zarr reader is the ability to parse JSON, to navigate a file system directory structure, and to inflate compressed blobs according to the indicated compression algorithm.

3.2.1 GeoZarr specification

GeoZarr[12] is a geospatial specification for Zarr. It leverages CF conventions to provide a means of georeferencing multidimensional arrays of geospatial observations. As of August 2023, he charter for a GeoZarr working group at OGC has been submitted for review [13].

3.2.2 XArray library

The popular N-Dimensional Array library, XArray, supports reading from and writing to Zarr stores.
A user can leverage the xarray.open_zarr method on an existing Zarr store and utilize the suite of XArray functionality to access and analyze the data. XArray will leverage the Zarr specification [2] and API to be performant when interfacing with distributed network storage, essentially abstracting away the underlying storage mechanism from the user.

3.2.3 DASK library

Zarr can be used with DASK to provide multi-threaded or multi-process parallelization utilizing lazy evaluation of arrays (ie. only accessing the parts of the arrays you need).

3.2.4 Intake library

Intake is a lightweight set of tools for loading and sharing data in data science projects. The intake-xarray library supports the Zarr data format.

3.2.5 Kerchunk library

Kerchunk allows you to access a variety of legacy scientific formats (including NetCDF and HDF5) as if they were a Zarr store, with the benefits Zarr provides for distributed network storage such as cloud object stores. This technique is attractive in that it does not require the reformatting and copying of the original archive to a Zarr store. Kerchunk works with the original format to provide to the user the functionality and performance of a Zarr store.
One limitation of Kerchunk is that the user is limited to the chunking scheme of the underlying data format which is usually driven by storage rather than access considerations. As such, Kerchunk's utility is very much dependent on the data it is used with.
It should be noted that Pangeo-Forge provides kerchunk support for those wishing to avoid replicating their data in a cloud-optimized format.

3.2.6 GDAL library

The GDAL [14] translation library provides a common API for working with a variety of raster data formats as well as tools for conversion between different raster formats. Zarr is a supported format through both GDAL's classic (2-dimensional) raster API and, since GDAL 3.1, through GDAL's multidimensional data API, which allows both direct analysis of Zarr archives using certain GDAL utilities and transformation between Zarr and other formats (like NetCDF or GeoTIFF). Support exists for version 2 of the specification with support for version 3 in an experimental phase.

3.2.7 NetCDF library

Zarr archives that follow specific metadata specifications — specifically, either the Xarray-Zarr specification or the NCZarr specification — can be opened and analyzed using the NetCDF C library.

4 Interoperability and Applicability Considerations

Zarr stores are most suitable for gridded data that is accessible via a distributed network. Non-gridded data, swath data for example, is less well-suited for replication or representation as a Zarr store. Zarr data stores of greater than around 20 MB in size are recommended.
EOSDIS' migration to the cloud makes a strong case for the proliferation of cloud-optimized data formats. The leading contenders in that field are Zarr, Cloud-optimized GeoTIFF and Parquet. Of those, Zarr is the most suitable for multi-dimensional data which constitutes the bulk

of the EOSDIS data archive. Cloud-optimized GeoTIFF does have applicability to EOSDIS data, but is out of scope for this RFC.
While we expect EOSDIS to produce Zarr data stores, we also expect the community at large to produce them as well, deriving them from EOSDIS archival data. One example of this is the Pangeo-Forge effort mentioned elsewhere in this document. A further driver for a more community-driven approach is that the considerations taken to produce a Zarr Store depend on the user consuming the data, specifically the chunking strategy. This means that one archived data set could have multiple Zarr stores tailored for different uses. This plays, somewhat, into the Pangeo-Forge philosophy that the generation of cloud-optimized data is not the sole domain of the data curator but also of the data scientist.
Zarr is a work in progress, particular in the realm of metadata standards. Attempts are in progress to mitigate this risk with proposals such as GeoZarr. Even so, there is a ground swell of adoption both within EOSDIS (see GES DISC work for Giovanni) and the earth science data community (see Pangeo-Forge).

5 Future versions of the specification

The current version of Zarr is version 2 [2] and that is the version endorsed by this recommended standard document. Version 3 of Zarr is currently under development. Version 3 will focus on the following,

- Feature parity and full interoperability across all major programming languages.
- Support for novel encoding technologies, storage technologies and features by a broader community.
- Reasonable performance characteristics of all Zarr implementations across a variety of different underlying storage technologies, including storage with high latency per operation.
- Improve performance for data with a very large number of chunks and/or with a variety of common access patterns.

6 References

6.1 Normative references

[1] Zarr homepage: https://zarr.dev
[2] Zarr specification: https://github.com/zarr-developers/zarr-specs

6.2 Informative references

[3] S3: https://aws.amazon.com/s3/
[4] DASK: https://www.dask.org/
[5] XArray: https://docs.xarray.dev/en/stable/
[6] Pangeo Forge: https://pangeo-forge.org/
[7] AWS open data registry: https://registry.opendata.aws
[8] MUR dataset on the AWS open data registry: https://registry.opendata.aws/mur/
[9] GIBS: https://wiki.earthdata.nasa.gov/display/GIBS/
[10] VEDA: https://www.earthdata.nasa.gov/esds/veda
[11] Zarr documentation: https://zarr.readthedocs.io/en/stable/

[12] GeoZarr: https://github.com/zarr-developers/geozarr-spec
[13] OGC GeoZarr Standards Working Group Charter: https://portal.ogc.org/files/105667 and https://eosdis.slack.com/archives/C04D3084GAZ/p1693497131582929
[14] GDAL: https://gdal.org/drivers/raster/zarr.html

7 Authors' Addresses

ESDIS Standards Office (ESCO)

Email: esco-staff@lists.nasa.gov

Web: https://earthdata.nasa.gov/esdis/esdis-standards-office-esco

Appendix A

Acronym	Description
API	Application Programming Interface
ARCO	Analysis-ready cloud optimized data
AWS	Amazon Web Services
CF	Climate and Forecast Metadata conventions
CMR	Common Metadata Repository
DOE	Department of Energy
EOSDIS	Earth Observing System Data and Information System
GES DISC	Goddard Earth Sciences Data and Information Services Center
GDAL	Geospatial Data Abstraction Library
GIBS	Global Imagery Browse Services
HDF	Hierarchical Data Format
NetCDF	Network Common Data Form
NOAA	National Oceanic and Atmospheric Administration
OGC	Open Geospatial Consortium
S3	Simple Storage Service
VEDA	Visualization, Exploration, and Data Analysis
WMS	OGC Web Mapping Service
WMTS	OGC Web Map Tiling Service

Review Questions

After reviewing this document, please rate the following questions on a scale of 1 to 5 where 1 = very unlikely, 2 = unlikely, 3 = neutral, 4 = likely, and 5 = very likely. (Detailed responses are also welcome.)

- How likely are you to support the endorsement of the Zarr storage specification version 2 for use in NASA Earth science data systems?
- How likely will this standard/convention be utilized by your organization?

Please answer the following demographic questions to help us ensure responses are received from target audiences and affiliations for this RFC.

- What is your role? Identify your job function such as data scientist, science communication & outreach, data management, operations, design & development, engineering, and/or any other descriptive terms.
- Where do you work? Provide name of DAAC, project, or organization which best represents your affiliation.