OGC® DOCUMENT: 21-032

External identifier of this OGC® document: http://www.opengis.net/doc/PER/t17-D047

Open Geospatial Consortium

OGC TESTBED 17: COG/ZARR EVALUATION ENGINEERING REPORT

ENGINEERING REPORT

PUBLISHED

Submission Date: 2021-11-19
Approval Date: 2021-12-17
Publication Date: 2022-01-24
Editor: Giovanni Giacco, Mauro Manente, Pedro Gonçalves, Martin Desruisseaux, Even Rouault

CONTENTS

LIST OF TABLES

LIST OF FIGURES

I ABSTRACT

The subject of this Engineering Report (ER) is the evaluation of Cloud Optimized GeoTIFF (COG) and Zarr data container implementations. The ER aims to:

- Describe the use cases adopted for the evaluation (with existing implementation and with Testbed-17 implementation);
- Identify the opportunity of proposing that COG and Zarr become OGC standards;
- Describe all components developed during the Testbed; and
- Provide an executive summary and a description of recommended future work items.

II EXECUTIVE SUMMARY

The focus of this is ER is documenting experiments in working with geospatial data in cloud-based environments. Specifically, the possibility of using two specific formats dedicated to managing the storage and distribution of images and data: Cloud Optimized GeoTIFF (COG) and Zarr.

The evaluation of the use of these two data structures is carried out in two independent contexts: the first one (named “Commercial Applications”) is based on the use of the current implementations of COG and Zarr within existing geospatial applications. The second one is related to the implementation work foreseen completed in Testbed 17. The latter context was named “T17” after the testbed. This was done in order to obtain an overview that represents the state of the art, the current development directions, and any future work. This context allows for better understanding if there are conditions for COG and Zarr to become OGC standards.

For both evaluation contexts, specific use cases were needed in order to give a central role to each of the two formats (or to both). The use cases had to be defined in such a way that measurable evaluation elements could be identified both to analyze the COG/Zarr implementation as a standalone element in the context and to perform (if possible) a comparison with other implementations or formats. Depending on the specific use cases, Key Performance Indicators (KPIs) were defined for the context and purpose of the evaluation and, where possible, comparative KPIs were identified.

For the Commercial Application context, the evaluation was performed on a cloud infrastructure implemented using virtual machines, storage systems, and networking functionalities that do not depend on the cloud service provider on which they are implemented. This model serves to make the evaluation as generic as possible given that comparing performance based on the cloud service provider is beyond the scope of this activity.

The evaluation in the T17 context refers to the actions carried out in the relevant development and/or testing environments by the teams that worked on the D180 (COG Implementation) and D181 (Zarr Implementation) deliverables.

On the basis of the above-mentioned approach, the evaluation provides:

- Comparison between the multidimensional COG implementation and ZARR;
- COG and ZARR integration with source catalogues in STAC format; and
- Overview on the Multidimensional Cloud Optimized GeoTIFF implementation with a specific programming language as an alternative to the GDAL drivers

This report also contains references to the documentation produced for both T17 and commercial application implementations.

III KEYWORDS

The following are keywords to be used by search engines and document catalogues.

ogcdoc, OGC document, COG, Zarr, Multidimensional COG, Cloud Optimized Geotiff

IV PREFACE

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. The Open Geospatial Consortium shall not be held responsible for identifying any or all such patent rights.

Recipients of this document are requested to submit, with their comments, notification of any relevant patent claims or other intellectual property rights of which they may be aware that might be infringed by any implementation of the standard set forth in this document, and to provide supporting documentation.

V SECURITY CONSIDERATIONS

No security considerations have been made for this document.

VI SUBMITTING ORGANIZATIONS

The following organizations submitted this Document to the Open Geospatial Consortium (OGC):

- Geomatys
- Latitudo 40
- Spatialys
- Terradue

VII SUBMITTERS

All questions regarding this document should be directed to the editor or the contributors (names in alphabetical order):

NAME	ORGANIZATION	ROLE
Martin Desruisseaux	Geomatys	Contributor
Giovanni Giacco	Latitudo 40	Contributor
Pedro Gonçalves	Terradue	Contributor
Mauro Manente	Latitudo 40	Editor
Even Rouault	Spatialys	Contributor

1

SCOPE

1 SCOPE

This OGC Testbed 17 Engineering Report represents deliverable D047 of the OGC Testbed 17 performed as part of the OGC Innovation Program.

This document aims to demonstrate the business value of the implementations for COG, with its extension to a multidimensional format, and for Zarr data container in the development environment used for this testbed and in the context of a commercial service.

2

TERMS, DEFINITIONS AND ABBREVIATED TERMS

2 TERMS, DEFINITIONS AND ABBREVIATED TERMS

This document uses the terms defined in OGC Policy Directive 49, which is based on the ISO/IEC Directives, Part 2, Rules for the structure and drafting of International Standards. In particular, the word “shall” (not “must”) is the verb form used to indicate a requirement to be strictly followed to conform to this document and OGC documents do not use the equivalent phrases in the ISO/IEC Directives, Part 2.

This document also uses terms defined in the OGC Standard for Modular specifications (OGC 08-131r3), also known as the ‘ModSpec’. The definitions of terms such as standard, specification, requirement, and conformance test are provided in the ModSpec.

For the purposes of this document, the following additional terms and definitions apply.

2.1. Terms and definitions

2.1.1. **Application**

A self-contained set of operations to be performed, typically to achieve a desired data manipulation, written in a specific language (e.g., Python, R, Java, C++, C#, IDL).

2.1.2. **Application Package**

A platform independent and self-contained representation of an Application, providing executables, metadata, and dependencies such that it can be deployed to and executed within an Exploitation Platform.

2.1.3. **Container**

A container is a standard unit of software that packages code and all its dependencies such that the container includes everything needed to run an application: code, runtime, system tools, system libraries, and settings.

2.2. Abbreviated terms

API	Application Programming Interface
CAMS	Copernicus Atmosphere Monitoring Service
COG	Cloud Optimized GeoTIFF
CRS	Coordinate Reference Systems
CWL	Common Workflow Language
EO	Earth Observation
ER	Engineering Report
GC	Garbage Collector
GDAL	Geospatial Data Abstraction Library
IFD	Image File Directory
OTB	Orfeo Toolbox
SIS	Spatial Information System
STAC	Spatio-Temporal Asset Catalog

3

INTRODUCTION

3 INTRODUCTION

This ER aims to provide an overview that represents the state of the art, the current development directions, and the future work that should be done in order to understand if there are conditions for COG and Zarr to become standards.

In order to achieve this goal, the evaluation activities alternated between the work of implementing a new driver for reading/writing a multidimensional version of COG files and that for managing the Zarr format in its version 2 as well as the use of these implementations both in the test/development context and in a cloud environment used for the provision of commercial services based on the processing of Earth observation data.

The evaluation of the aforementioned new implementations in the testing context and in the commercial context was carried out using qualitative elements and comparisons with existing implementations for the former and trying to identify quantitative and qualitative KPIs for the latter. This choice is justified by the impossibility of carrying out a quantitative verification that would be independent of the specific use case. However, seeing the opportunity to verify the application of the COG and Zarr implementations in an operational context on specific use cases, it was decided to differentiate the evaluation methodologies.

This document thus provides descriptions for:

- The implementation of a driver for reading/writing multidimensional COG files as an extension of the one already present in the GDAL library;
- The implementation of a driver for managing the Zarr data container in its version 2 and an evaluation of the Zarr V.3 specification;
- The evaluation of the multidimensional COG driver on a specific use case in which the possibility of using a STAC catalogue as a source for earth observation data to write a multidimensional COG file representing a multi-temporal analysis of an area of interest;
- An evaluation of current COG and Zarr implementations in the context of providing commercial urban monitoring services through a cloud-based infrastructure, together with an assessment of the use of implementations built in the testbed to satisfy the same use case; and
- An examination of the possibility of implementing a version of the multidimensional COG driver outside the GDAL context and using a specific programming language.

3.1. COG Format overview

COGs are standard GeoTIFFs. The COG file format is an extension of the GeoTIFF file format which enables more flexible access to geospatial data. COG supports users being able to perform HTTP range GET requests to extract specific portions of the file's data. COG relies on two complementary pieces of technology:

- first, the ability of GeoTIFFs to store not just the raw pixels of the image, but to organize those pixels in particular ways; and
- second, HTTP GET range requests that allows clients to ask for just the portions of a file that they need.

Using the first capability organizes the GeoTIFF so the latter's requests can easily select the parts of the file that are useful for processing.

COGs have an internal structure specified, but not enforced, by the GeoTIFF standard. The two main organizational techniques that COGs use are Tiling and Overviews. Further, the data is also compressed for more efficient transmittal.

- **Internal Tiling** creates a number of internal 'tiles' inside the actual image. This is instead of using simple 'stripes' of data. With a stripe of data, the whole file needs to be read to get the required piece. With tiles, much faster access to a specific area is possible. Therefore, only just the portion of the file that needs to be read is accessed.
- **Internal Overviews** create downsampled versions of the same image. This means it is 'zoomed out' from the original image — it has much less detail, but is also much smaller. Often a single GeoTIFF will have many overviews that match different zoom levels. These add size to the overall file, but are able to be served much faster, since the renderer just has to return the values in the overview.

Examples of the operations on GeoTIFFs converted to COGs in the cloud include:

- Pixel interleaving;
- Internal tiling with block size 256×256 pixels;
- Internal overviews with block size 128×128 pixels and downsampling levels of 2, 4, 8, 16, 32, and 64; and
- Compression with the "deflate" algorithm.

The internal structure of COGs promotes efficient consumption of the format in networked environments. This means that relatively few accesses of a portion of a COG using HTTP GET range requests can be used to extract metadata, overview imagery, a spatial subset, and/or a single multispectral band. COGs also define a structure for a pyramid of overview tiles, enabling the possibility of low-bandwidth preview before access. The COG specification can be found at https://github.com/cogeotiff/cog-spec/blob/master/spec.md.

3.2. ZARR Format overview

Multidimensional array data (i.e., N-dimensional arrays) are ubiquitous in scientific research and engineering. In GDAL 3.1, a multidimensional data model, and a corresponding API, was added. The multidimensional raster API is a generalization of the traditional 2D Raster Data Model, designed to address 3D, 4D, or higher dimension datasets. The most common format for relevant datasets is an archive of hundreds to thousands of individual netCDF / HDF files. This format does not work particularly well on cloud object storage; the opaque nature of the files makes it nearly impossible to extract a single variable or piece of data or metadata without reading the whole file. Zarr is the result of an effort to overcome these issues.

Zarr is an open-source specification for the storage of ND-arrays and associated metadata. Zarr is an array-based hierarchical data store, conceptually similar to NetCDF-4 in terms of its ability to capture and express metadata and data.

The major advantages to the Zarr format are:

- Metadata is kept separate from data in a lightweight .json format;
- Arrays are stored in a flexible chunked / compressed binary format; and
- Individual chunks can be retrieved independently in a thread-safe manner.

Zarr stores hierarchies as “logical paths” which, by default, expand to separate keys in object stores such as Amazon S3. As such a single conceptual file may in fact be several files on disk or in object stores. This can improve both parallelism and granularity in access. Metadata are stored into JSON text files.

4

MULTIDIMENSIONAL CLOUD OPTIMIZED GEOTIFF

4 MULTIDIMENSIONAL CLOUD OPTIMIZED GEOTIFF

4.1. Introduction to multidimensional COG file format

As a profile of TIFF (or BigTIFF) and GeoTIFF, the COG format, logically inherits their limitations, and in particular the fact that TIFF was designed for 2D images. There is no standardized way of extending TIFF to more dimensions, as the size and block size of each dimension relies on are stored in a dedicated TIFF tag: ImageLength, ImageWidth, TileLength, TileWidth. One could possibly imagine newer keys to define the characteristics of the new dimension, but such a file would be completely incompatible with existing readers. This would also be contrary to the philosophy of the COG format to use the degree of freedoms of the TIFF format to their best extent, while keeping compliant with the specification.

Another possibility for storing multidimensional content would have been to use the multiple sample/bands capability of the TIFF format for each pixel value. Each extra dimension could have been modeled as an additional sample. One drawback of that method is that it may restrict use of some compression methods (such as JPEG) that only work, or are mostly optimized, to work with a given restricted number of components. It would also have required providing some metadata to explain the interleaving scheme between the sample/band addressing space and multi-dimensional addressing space.

TIFF files can contain a sequence of images, each called an Image File Directory (IFD). This is the mechanism typically used in geospatial applications to encode an image pyramid, with lower resolutions of the full resolution image, or to provide a validity mask, when an alpha channel cannot be used. In this testbed, the participants have experimented with using one IFD for each non-2D slice. For example, for a 3D array with a T dimension with 4 items and a Z dimension with 10 items, the resulting COG files will have 4*10=40 IFDs. Because existing TIFF readers that can handle several IFDs and can read such a dataset correctly, this solution is probably the one that has the best backward compatibility property.

The philosophy “all headers first” of the COG format still applies: to limit seeking within the file, all IFDs descriptors (fields and their values) should be put at the beginning of the file.

Compression codecs (Deflate, LZW, ZSTD, Lerc, etc.) usable for standard TIFF / COG can be used for multidimensional COG.

Two dimensional TIFF files have a built-in chunking mechanism, using the concept of rectangular tiles, to be able to use portions of an image in an efficient way. For multidimensional COG, the participants used that mechanism for the two fastest varying dimensions (generally the horizontal spatial ones) and extended it for the upper dimensions in a looser way. The chunk sizes that relates to the upper dimensions is an information hint on how to interleave 2D tiles in the file.

Let us consider a 3D 4×256x512 array with the following characteristics

- X size: 512, chunked by 256
- Y size: 256, chunked by 256
- Z size: 4, chunked by 2

The content is organized into four chunks:

- One with samples Z=0..1, Y=0..255, X=0..255, which will be physically split in two tiles: one tile belonging to IFD 0 (for Z=0), and another one belonging to IFD 1 (for Z=1)
- One with samples Z=0..1, Y=0..255, X=256..511, which will be physically split in two tiles: one tile belonging to IFD 0 (for Z=0), and another one belonging to IFD 1 (for Z=1)
- One with samples Z=2..3, Y=0..255, X=0..255, which will be physically split in two tiles: one tile belonging to IFD 2 (for Z=2), and another one belonging to IFD 3 (for Z=3)
- One with samples Z=2..3, Y=0..255, X=256..511, which will be physically split in two tiles: one tile belonging to IFD 2 (for Z=2), and another one belonging to IFD 3 (for Z=3)

The order of chunks themselves in the file could potentially be "random," with the constraint that tiles belonging to a same chunk should be stored sequentially in the file to obtain the desired effect. However, the GDAL driver uses a "row-order" logic, generalizing to ND the logic applied for 2D COGs.

The physical layout of the file is, from its start to its end:

- IFD 0: descriptor for Z=0
- IFD 1: descriptor for Z=1
- IFD 2: descriptor for Z=2
- IFD 3: descriptor for Z=3
- Tile offset/bytecount tables of IFD 0
- Tile offset/bytecount tables of IFD 1
- Tile offset/bytecount tables of IFD 2
- Tile offset/bytecount tables of IFD 3
- Tile data for Z=0, Y=0..255, X=0..255
- Tile data for Z=1, Y=0..255, X=0..255

- Tile data for Z=0, Y=0..255, X=256..511
- Tile data for Z=1, Y=0..255, X=256..511
- Tile data for Z=2, Y=0..255, X=0..255
- Tile data for Z=3, Y=0..255, X=0..255
- Tile data for Z=2, Y=0..255, X=256..511
- Tile data for Z=3, Y=0..255, X=256..511

The description of the shape, dimensions and chunk size of a multidimensional GeoTIFF array are stored as key/value pairs in the GDAL_METADATA TIFF tag (https://www.awaresystems.be/imaging/tiff/tifftags/gdal_metadata.html). The first IFD contains a full description, which is sufficient for a reader to expose all metadata on the dataset. For the sake of compactness, following IFDs can contain only a subset of it, describing the values of the dimension axis beyond the 2 first dimensions (this is a provision for GeoTIFF readers unaware of the design to still be able to make some sense of it).

Example of metadata for the first IFD of a 3D array:

```
<GDALMetadata>
  <Item name="VARIABLE_NAME">myarray</Item>
  <Item name="DIMENSION_0_NAME">dimZ</Item>
  <Item name="DIMENSION_0_SIZE">5</Item>
  <Item name="DIMENSION_0_BLOCK_SIZE">2</Item>
  <Item name="DIMENSION_0_TYPE">a</Item>
  <Item name="DIMENSION_0_DIRECTION">b</Item>
  <Item name="DIMENSION_0_IDX">0</Item>
  <Item name="DIMENSION_0_DATATYPE">Int32</Item>
  <Item name="DIMENSION_0_VALUES">1,2,3,4,5</Item>
  <Item name="DIMENSION_0_VAL">1</Item>
  <Item name="DIMENSION_1_NAME">dimY</Item>
  <Item name="DIMENSION_1_SIZE">257</Item>
  <Item name="DIMENSION_1_BLOCK_SIZE">256</Item>
  <Item name="DIMENSION_2_NAME">dimX</Item>
  <Item name="DIMENSION_2_SIZE">280</Item>
  <Item name="DIMENSION_2_BLOCK_SIZE">256</Item>
</GDALMetadata>
```

Example of metadata for the second IFD of a 3D array:

```
<GDALMetadata>
    <Item name="VARIABLE_NAME">myarray</Item>
    <Item name="DIMENSION_0_NAME">dimZ</Item>
    <Item name="DIMENSION_0_IDX">1</Item>
    <Item name="DIMENSION_0_VAL">2</Item>
</GDALMetadata>
```

4.2. Specification

Currently at https://github.com/opengeospatial/Multidimensional-TIFF

4.3. GDAL implementation

The source code for the implementation is available in https://github.com/rouault/gdal/tree/mdim_cog (corresponding to a development version of GDAL 3.4.0), and its documentation in https://github.com/rouault/gdal/blob/mdim_cog/gdal/doc/source/drivers/raster/gtiff_multidimensional.rst.

The GeoTIFF driver has been extended to implement the GDAL multidimension API for reading and writing. When reading, any area of interest, potentially crossing several chunks, can be requested. The driver has optimization to request in a single network request chunks that span over several contiguous TIFF tiles.

For writing, the area of interest should be aligned on the boundaries of whole chunks. This is to increase implementation simplicity, and also to avoid recompressing chunks previously partly written, which could cause them to not fit in their original location.

The implementation is limited currently to one sample/band per array (SamplePerPixel = 1). This is not a limitation of the design of the multidimensional COG format, but only an implementation one in the context of this testbed.

4.4. Network accesses

- At least one HTTP GET request with a Range fetching the first kilobytes of the file is sufficient to get the first IFD which provides all the metadata on the array.
- To be able to read a chunk that intersect the last IFDs, a GET request fetching the first hundred of kilobytes will be needed to get all the IFD descriptors.
- Reading a chunk requires knowing the offset and size of the tiles it is made of. The corresponding parts of the TileOffsets and TileByteCount arrays must thus be read.
- Finally, a HTTP GET request with a Range that spans over the tiles comprising the chunk of interest must be emitted.

For a long living process, the initial cost of reading the IFDs and the TileOffsets/TileByteCount arrays can be amortized, and the amortized cost is one HTTP GET request per chunk.

4.5. Limitations of the design

The main limitation of the approach is linked to the fact that there must be one IFD for the product of samples along the upper dimensions.

There is no mechanism in TIFF to quickly access a given IFD without looking sequentially at the chain of linked IFDs. Note: one could potentially imagine adding an index table, in a “ghost” area of a file to do that, with the caveat that writers non aware of it would not update it, causing confusion to readers using out-of-sync data.

Somewhat linked to the above limitation, the libtiff implementation of TIFF used by GDAL, has a limitation to 65,536 IFDs.

The TIFF file format uses 32-bit offsets and as such, is limited to 4 gigabytes. However, multidimensional files can reach more quickly the 4 GB file size limit, hence handling BigTIFF is a quasi-requirement for multidimensional COG files. BigTIFF is a new variant of TIFF, that closely resembles TIFF, but uses 64-bit offsets instead, breaking the 4 GB boundary.

Many IFDs also imply a potential significant overhead to store the values of the required tag for an IFD, which can be significant for the TileOffsets/TileByteCount arrays if the spatial extent of the file is large.

All-in-all, the number of samples along the upper dimensions should remain relatively small, on the order of a few thousands maximum, for the design to remain practical.

The current implementation is also limited to a single multidimensional array per file. This restriction could potentially be waved by deciding on a convention on how to order IFDs belonging to different arrays. However, the above mentioned limits would then be triggered.

4.6. Comparison of Zarr versus multidimensional COG file

- The main difference is that Zarr uses a multiple file approach (metadata files + one file per data chunk) whereas GeoTIFF uses a single file approach. A single file approach is practical for the sake of moving files, but is less convenient when updates must be made (“holes” can appear).
- Zarr has been designed from the start for multidimensional use. The multidimensional COG approach stretches the original design of the TIFF format quite far beyond its grounds.
- Multidimensional COG inherits the properties of TIFF and GeoTIFF, in particular a standardized way of encoding 2D spatial referencing, which Zarr has natively.
- Both formats can be difficult to implement in an exhaustive way, given they have each an important set of compression codecs, that are not all formally defined in their respective core specifications.

5

SPATIAL REFERENCING IMPLEMENTATION IN ZARR

5 SPATIAL REFERENCING IMPLEMENTATION IN ZARR

As Zarr uptake is growing, there has been increasing interest in having support in the GDAL (Geospatial Data Abstraction Library) open source library, providing read and write access. This is in addition to the many other raster formats handled by GDAL. The GDAL unified API (Application Programming Interface) to many formats make it appealing for application developers that want to develop a library or application being able to address different file formats, without having to handle the low-level details of each format.

Zarr was designed taking inspiration from the netCDF and HDF5 API and data models, offering a hierarchical structure of content (groups), multidimensional arrays and attributes. Two command line utilities, gdalmdiminfo and gdalmdimtranslate, are also available to respectively provide information on the content of a multidimensional dataset (as a JSON formatted report), and to convert between formats, with options to select arrays of interest and perform subsetting. Prior to Testbed-17, multidimensional support was available in the drivers for the netCDF, HDF4, HDF5, GRIB, VRT (GDAL Virtual Dataset), and in-memory drivers.

5.1. Driver capabilities

The Zarr driver can be pointed to open any part of a Zarr V2 hierarchy, either at the leaf (array) level or at the group level. By default, it will limit not automatically recurse into the hierarchy, but will do that on-demand when using the GetGroupNames() API.

Regarding arrays, the driver handles the following aspects of the specification: - support for any number of dimensions - support for up to 16e18 samples (ie that fits on 64-bit) in each dimension - support for chunks of an arbitrary size, provided that it fits into RAM - support for most commonly found datatypes, such as numeric datatypes (both endianness supported) or strings; Compound data types are also supported - support for Fortran or C element ordering in chunks - support for the null value - support for common compression methods: ZLib, GZip, BZ4, ZSTD, LZMA and Blosc - support for the Delta filter methods (filters are somewhat similar to compression methods, except they generally not contribute directly to reducing the chunk size) - support for attributes: numeric (integer, real), string, boolean data types, and 1D arrays of the previous types are mapped to the corresponding GDAL data type. As the value of a Zarr attribute can be any valid JSON datatype, more complex values, such as arrays with 2 dimensions or more, or with heterogeneous data types for their member, or using JSON object, are exposed as a JSON encoded string with a hint that it is JSON content. For example this enables lossless conversion when doing Zarr to Zarr conversion.

The driver uses the GDAL Virtual File System abstraction to access both metadata as JSON files or data files consisting of chunks. This means that Zarr datasets can be on the local file system, in a .zip, .tar or .tgz archive, in-memory, or remote using generic HTTP access or one of the dedicated (AWS S3, Google Cloud Storage, Microsoft Azure Blobs, Alibaba Cloud OSS, OpenStack Swift, Hadoop File system) storage, or potentially a combination of those (e.g., a .zip file on AWS S3).

The implementation has the following limitations. - Unhandled datatypes are timedelta, datetime, Unicode, and 'void', or non-scalar members of compound datatypes. Some of those limitations are linked to limitations in the GDAL abstract model itself. - Other compression methods than the ones mentioned above are not supported. However, GDAL offers an API to plug additional compression methods if needed. The issue with compression methods is that the Zarr V2 specification itself does not specify them. The compression methods are informally expected to be the ones provided by the https://github.com/zarr-developers/numcodecs library used by the Python Zarr reference implementation. There are similar issues with filter methods: None are specified in the specification, and the ones of the numcodecs library are the de-facto standard

As the GDAL multidimensional API is rather new and not yet heavily used, the driver also implements a bridge to the classic 2D raster API for reading or writing. For 2D-only datasets, the bridge is natural. For 3D or more datasets, the driver will expose each 2D slice as a GDAL sub-dataset.

The documentation of the Zarr driver can be found at https://gdal.org/drivers/raster/zarr.html. It will be available in the official GDAL 3.4.0 release, planned for November 2021.

5.2. Spatial referencing

The Zarr specification makes no provision for the spatial referencing of geospatial data. The spatial referencing comes in two parts.

- Being able to relate a grid point to a (X,Y) or (longitude,latitude) coordinate value. In netCDF, this is typically done by having a numeric 1D-array holding the correspondence between sample index and georeferenced ordinate and this is approach is also commonly found in Zarr datasets converted from netCDF. The GDAL Zarr driver has special identification of numeric 1D-array to assign them as indexing variables of a dimension.
- Assigning a coordinate reference system (CRS), whose conceptual method is for example described by Topic 2 of the OGC Abstract Specification, with different possible encodings. There is no clear "market-approved" solution currently in Zarr to describe the CRS. If data is directly transformed from netCDF files following the netCDF CF conventions (https://cfconventions.org/), the referencing of the resulting Zarr could also be interpreted in a similar way as in a netCDF reader. This, however, assumes that the interpretation of the netCDF CF encoding of CRS is done in a way that is sufficiently un-entangled from the netCDF API, which is not currently the case in the GDAL implementation. Furthermore, the design of the netCDF CF CRS encoding uses an extra netCDF variable with no content other than attributes, which if translated directly into a Zarr dataset, results in a not very elegant solution, since the reader has to know how to relate

the array that contains the data of interest with the somewhat dummy array that holds the CRS information.

The testbed participants decided to experiment with a cleaner and novel solution given the design of the Zarr JSON files by using a _CRS attribute. Its value is an object with the potential keys:

- *wkt* (the value is then a string using a WKT:2019 encoding);
- *url* (the value is then a string with a OGC URL); and
- *projjson* (the value is then a JSON object following the PROJJSON encoding, https://proj.org/specifications/projjson.html, which is a JSON encoding with similar capabilities as WKT2:2019).

One or several of those keys can be present. The rules of precedence applied on reading by the Zarr driver are: it will use *url* by default, if not found will fallback to *wkt* and then *projjson*.

Below is an example of a *.attrs* file containing the CRS definition for the “NAD27 / UTM zone 11N” projected CRS.

```
{
  "_CRS":{
    "wkt":"PROJCRS[\"NAD27 / UTM zone 11N\",BASEGEOGCRS[\"NAD27\",DATUM[\"North
 American Datum 1927\",ELLIPSOID[\"Clarke 1866\",6378206.
4,294.978698213898,LENGTHUNIT[\"metre\",1]]],PRIMEM[\"Greenwich
\",0,ANGLEUNIT[\"degree\",0.0174532925199433]],ID[\"EPSG
\",4267]],CONVERSION[\"UTM zone 11N\",METHOD[\"Transverse
 Mercator\",ID[\"EPSG\",9807]],PARAMETER[\"Latitude of natural
 origin\",0,ANGLEUNIT[\"degree\",0.0174532925199433],ID[\"EPSG
\",8801]],PARAMETER[\"Longitude of natural origin\",-117,ANGLEUNIT[\"degree
\",0.0174532925199433],ID[\"EPSG\",8802]],PARAMETER[\"Scale factor
 at natural origin\",0.9996,SCALEUNIT[\"unity\",1],ID[\"EPSG
\",8805]],PARAMETER[\"False easting\",500000,LENGTHUNIT[\"metre\",1],ID[\"EPSG
\",8806]],PARAMETER[\"False northing\",0,LENGTHUNIT[\"metre\",1],ID[\"EPSG
\",8807]]],CS[Cartesian,2],AXIS[\"easting\",east,ORDER[1],LENGTHUNIT[\"metre
\",1]],AXIS[\"northing\",north,ORDER[2],LENGTHUNIT[\"metre\",1]],ID[\"EPSG
\",26711]]",

    "projjson":{
      "$schema":"https://proj.org/schemas/v0.3/projjson.schema.json",
      "type":"ProjectedCRS",
      "name":"NAD27 / UTM zone 11N",
      "base_crs":{
        "name":"NAD27",
        "datum":{
          "type":"GeodeticReferenceFrame",
          "name":"North American Datum 1927",
          "ellipsoid":{
            "name":"Clarke 1866",
            "semi_major_axis":6378206.4,
            "inverse_flattening":294.978698213898
          }
        },
        "coordinate_system":{
          "subtype":"ellipsoidal",
          "axis":[
            {
              "name":"Geodetic latitude",
              "abbreviation":"Lat",
              "direction":"north",
              "unit":"degree"
            },
            {
              "name":"Geodetic longitude",
              "abbreviation":"Lon",
              "direction":"east",
              "unit":"degree"
            }
          ]
        },
        "id":{
          "authority":"EPSG",
          "code":4267
        }
      },
      "conversion":{
        "name":"UTM zone 11N",
        "method":{
          "name":"Transverse Mercator",
          "id":{
            "authority":"EPSG",
            "code":9807
          }
        },
        "parameters":[
          {
            "name":"Latitude of natural origin",
            "value":0,
            "unit":"degree",
            "id":{
              "authority":"EPSG",
              "code":8801
            }
          },
          {
            "name":"Longitude of natural origin",
            "value":-117,
            "unit":"degree",
            "id":{
              "authority":"EPSG",
              "code":8802
            }
          },
          {
            "name":"Scale factor at natural origin",
            "value":0.9996,
            "unit":"unity",
            "id":{
              "authority":"EPSG",
              "code":8805
            }
          },
          {
            "name":"False easting",
            "value":500000,
            "unit":"metre",
            "id":{
              "authority":"EPSG",
              "code":8806
```

```
            }
          },
          {
            "name":"False northing",
            "value":0,
            "unit":"metre",
            "id":{
              "authority":"EPSG",
              "code":8807
            }
          }
        ]
      },
      "coordinate_system":{
        "subtype":"Cartesian",
        "axis":[
          {
            "name":"Easting",
            "abbreviation":"",
            "direction":"east",
            "unit":"metre"
          },
          {
            "name":"Northing",
            "abbreviation":"",
            "direction":"north",
            "unit":"metre"
          }
        ]
      },
      "id":{
        "authority":"EPSG",
        "code":26711
      }
    },

    "url":"http://www.opengis.net/def/crs/EPSG/0/26711"
  }
}
```

5.3. Multi-threading

Given the design of the format, it is quite natural to use multi-threading capabilities to fetch and decode in parallel chunks that would intersect an area of interest. The GDAL Zarr driver implements a AdviseRead() method that can invoked by the user to express its area of interest and do parallel prefetching and decoding. The content is then cached into RAM, and further calls using the Read() method that intersects the prefetched area of interest will directly use the cache.

5.4. Extensions implemented

- The GDAL Zarr driver implements in reading and writing the *ARRAY_DIMENSIONS_ attribute coming from the XArray Zarr implementation: http://xarray.pydata.org/en/stable/internals/zarr-encoding-spec.html. This extension addresses a limitation of the core Zarr specification, which is that dimensions of an array are unnamed. Only their size (number of samples along the dimension) is specified. The value of the* ARRAY_DIMENSIONS_ attribute is a JSON array with the names of each dimension.

e.g.,

```
{
    "__ARRAY_DIMENSIONS_": ["z", "lat", "lon"]
}
```

This extension is needed to be able to implement the concept of indexing variable and relate a dimension to the values taken along that axis.

- The GDAL Zarr driver implements support for reading and writing through the "Consolidated metadata" extension. This extension is not specified in the Zarr core specification, but is implemented by the Python reference implementation. The extension consists of a single .zmetadata file, located at the root of a Zarr dataset, that is a JSON file which gathers the content of all *.zgroup*, *.zarray* and *.zattrs* JSON files located in the dataset. The .zmetadata file is especially useful when the Zarr dataset is hosted on network storage. Network storage has a rather high-latency access, which is detrimental for metadata discovering. When the network file storage does not offer a way of listing file content, automatic exploration of the hierarchy would also be impossible without the .zmetadata file. The downside of this extension is that a Zarr implementation not aware of the extension and also has update capabilities would not update it. This the .zmetadata file would go out of sync, causing issues for readers.

- The GDAL Zarr driver implements, in reading only the NCZarr, version 2, extension: https://www.unidata.ucar.edu/software/netcdf/documentation/NUG/nczarr_head.html. NCZarr originates from an optional mode in the Zarr implementation of the the netCDF library to fully support all concepts of the netCDF data model. Original netCDF data types are thus included in additional metadata. But the main interest is that the dimension concept is fully supported. The main enhancement over the *_ARRAY_DIMENSIONS* extension is the ability to reference dimensions that are not contained in the same level of the hierarchy than the considered array.

5.5. Network accesses

For a Zarr dataset hosted on a network storage, the following requests will be performed.

- A HTTP GET request to retrieve the .zmetadata file, if present. When the file is found, then the complete hierarchy of groups, arrays and their attributes is immediately available. However for complex datasets, the .zmetadata file can be several megabytes or more in size. Therefore, reading the file would not be the best strategy if reading, for example ,one chunk at a time from a process that is restarted each time.
- A HTTP GET request to access each chunk of data. Of course, depending on its access pattern the shape of the chunk might influence the performance of the processing. If using a <time, long, lat> array and doing computations where access to several samples along time is not needed, then a small value (1 for example) for the size of the chunk along the time dimension is appropriate. On the contrary, if computing indicators that require aggregating several values over time for a given spatial location, a larger value will be more appropriate. There is a compromise to do on the resulting chunk size. If too large, and only processing of small portions of the dataset is needed (worst case here is extraction of a single value in the array), then useless network access and CPU time will be consumed. If too small, a larger number of GET requests will be issues, with latency issues and larger costs (for storage where there is a cost assigned to each network request).

Network access triggered by GDAL can be monitored by setting the *CPL_VSIL_NETWORK_STATS_ENABLED* environment variable to *YES*. At the end of the process running GDAL, a report will be emitted with the number and nature of network requests, the number of bytes transferred and the objects accessed.

One should note that for Zarr, no HTTP range request is needed (metadata or data files are read in their entirety), contrary to a COG file, which simplifies a bit a client implementation.

5.6. Zarr V3

A new version of the Zarr specification has been proposed by a group of developers: https://zarr-specs.readthedocs.io/en/core-protocol-v3.0-dev/protocol/core/v3.0.html.

The GDAL Zarr driver implements this version in reading and writing (on writing, the user has to explicitly select the version through a dedicated *FORMAT* creation option, as the default version is v2).

Below are quotes of the comparison of the differences, https://zarr-specs.readthedocs.io/en/core-protocol-v3.0-dev/protocol/core/v3.0.html#comparison-with-zarr-v2, between v2 and v3 of the specification.

- In v3 each hierarchy has an explicit root and must be opened at the root. In v2 there was no explicit root and a hierarchy could be opened at its original root or at any sub-group.
- In v3 by using different prefixes, the storage keys have been redesigned to separate the space of keys used for metadata and data. This is intended to support more performant listing and querying of metadata documents on high latency stores. There are also differences including a change to the default separator used to construct chunk keys and the addition of a key suffix for metadata keys.
- v3 has explicit support for protocol extensions via defined extension points and mechanisms.
- v3 allows for greater flexibility in how groups and arrays are created. In particular, v3 supports implicit groups, which are groups that do not have a metadata document but whose existence is implied by descendant nodes. This change enables multiple arrays to be created in parallel without generating any race conditions for creating parent groups.
- The set of data types specified in v3 is less than in v2. Additional data types will be defined via protocol extensions.

The testbed participants concur with the analysis, with the following observations.

- The separation of metadata and data should solve issues when listing recursively the content of a Zarr dataset to explore its content. In Zarr V2, with big datasets comprising of millions of tiles, directory listing can quickly become impractical. This issue was experienced during development of the driver, which caused revision of the strategy to decide if proactive of directory content or a more on-demand strategy must be implemented.
- The naming strategy of files in Zarr V3 is more complex than Zarr V2, particularly with differences between the content at the root of the dataset or in a group. This is mostly a detail that matters to implementers of the specification and is transparent to users.
- The concept of an implicit vs explicit group represents a complication compared to the v2 specification. It would have probably been better to decide for one solution. The GDAL Zarr driver always generates groups with an explicit metadata document.
- The restriction of data types in the core specification is a welcome one, as it removes a lot of complexity that is needed only for somewhat esoteric use cases.
- Zarr V3 compression schemes are specified formally. Currently only the GZip method has been specified.

There does not seem to be any widespread use of the Zarr V3 specification in publicly accessible datasets. One of the main reasons is probably that the Python Zarr reference implementation does not support it (at time of writing).

5.7. Examples

Metadata related to the Multi-Scale Ultra High Resolution (MUR) Sea Surface Temperature (SST) dataset (https://registry.opendata.aws/mur/) can be retrieved with

```
gdalmdiminfo /vsis3/mur-sst/zarr-v1 --config AWS_NO_SIGN_REQUEST YES
```

Figure 1

which outputs:

```
{
  "type": "group",
  "driver": "Zarr",
  "name": "/",
  "attributes": {
    "Conventions": "CF-1.7",
    "Metadata_Conventions": "Unidata Observation Dataset v1.0",
    "acknowledgment": "Please acknowledge the use of these data with the
 following statement:  These data were provided by JPL under support by NASA
 MEaSUREs program.",
    "cdm_data_type": "grid",
    "comment": "MUR = \"Multi-scale Ultra-high Resolution\"",
    "creator_email": "ghrsst@podaac.jpl.nasa.gov",
    "creator_name": "JPL MUR SST project",
    "creator_url": "http://mur.jpl.nasa.gov",
    "date_created": "20200124T010755Z",
    "easternmost_longitude": 180,
    "file_quality_level": 3,
    "gds_version_id": "2.0",
    "geospatial_lat_resolution": 0.00999999977648258209,
    "geospatial_lat_units": "degrees north",
    "geospatial_lon_resolution": 0.00999999977648258209,
    "geospatial_lon_units": "degrees east",
    "history": "created at nominal 4-day latency; replaced nrt (1-day latency)
 version.",
    "id": "MUR-JPL-L4-GLOB-v04.1",
    "institution": "Jet Propulsion Laboratory",
    "keywords": "Oceans > Ocean Temperature > Sea Surface Temperature",
    "keywords_vocabulary": "NASA Global Change Master Directory (GCMD) Science
 Keywords",
    "license": "These data are available free of charge under data policy of
 JPL PO.DAAC.",
    "metadata_link": "http://podaac.jpl.nasa.gov/ws/metadata/dataset/?format=
iso&shortName=MUR-JPL-L4-GLOB-v04.1",
    "naming_authority": "org.ghrsst",
    "netcdf_version_id": "4.1",
    "northernmost_latitude": 90,
    "platform": "Terra, Aqua, GCOM-W, MetOp-A, MetOp-B, Buoys/Ships",
    "processing_level": "L4",
    "product_version": "04.1",
    "project": "NASA Making Earth Science Data Records for Use in Research
 Environments (MEaSUREs) Program",
    "publisher_email": "ghrsst-po@nceo.ac.uk",
    "publisher_name": "GHRSST Project Office",
    "publisher_url": "http://www.ghrsst.org",
    "references": "http://podaac.jpl.nasa.gov/Multi-scale_Ultra-high_
Resolution_MUR-SST",
    "sensor": "MODIS, AMSR2, AVHRR, in-situ",
    "source": "MODIS_T-JPL, MODIS_A-JPL, AMSR2-REMSS, AVHRRMTA_G-NAVO,
 AVHRRMTB_G-NAVO, iQUAM-NOAA/NESDIS, Ice_Conc-OSISAF",
    "southernmost_latitude": -90,
    "spatial_resolution": "0.01 degrees",
    "standard_name_vocabulary": "NetCDF Climate and Forecast (CF) Metadata
 Convention",
    "start_time": "20200116T090000Z",
    "stop_time": "20200116T090000Z",
    "summary": "A merged, multi-sensor L4 Foundation SST analysis product from
 JPL.",
    "time_coverage_end": "20200116T210000Z",
    "time_coverage_start": "20200115T210000Z",
    "title": "Daily MUR SST, Final product",
    "uuid": "27665bc0-d5fc-11e1-9b23-0800200c9a66",
    "westernmost_longitude": -180
  },
  "dimensions": [
    {
      "name": "lat",
      "full_name": "/lat",
      "size": 17999,
      "type": "HORIZONTAL_Y",
      "direction": "NORTH",
      "indexing_variable": "/lat"
    },
    {
      "name": "lon",
      "full_name": "/lon",
      "size": 36000,
      "type": "HORIZONTAL_X",
      "direction": "EAST",
      "indexing_variable": "/lon"
    },
    {
      "name": "time",
      "full_name": "/time",
      "size": 6443,
      "type": "TEMPORAL",
      "indexing_variable": "/time"
    }
  ],
  "arrays": {
    "lat": {
      "datatype": "Float32",
      "dimensions": [
        "/lat"
      ],
      "dimension_size": [
        17999
      ],
      "block_size": [
        17999
      ],
      "attributes": {
        "axis": "Y",
        "comment": "none",
        "long_name": "latitude",
        "valid_max": 90,
        "valid_min": -90
      },
      "unit": "degrees_north",
      "nodata_value": "NaN"
    },
    "lon": {
      "datatype": "Float32",
      "dimensions": [
        "/lon"
      ],
      "dimension_size": [
        36000
      ],
      "block_size": [
        36000
      ],
      "attributes": {
        "axis": "X",
        "comment": "none",
        "long_name": "longitude",
        "valid_max": 180,
        "valid_min": -180
      },
      "unit": "degrees_east",
      "nodata_value": "NaN"
    },
    "time": {
      "datatype": "Float64",
      "dimensions": [
        "/time"
      ],
      "dimension_size": [
        6443
      ],
      "block_size": [
        5
      ],
      "attributes": {
        "axis": "T",
        "calendar": "proleptic_gregorian",
        "comment": "Nominal time of analyzed fields",
        "long_name": "reference time of sst field"
      },
      "unit": "days since 2002-06-01 09:00:00"
    },
    "analysed_sst": {
      "datatype": "Int16",
      "dimensions": [
        "/time",
        "/lat",
        "/lon"
      ],
      "dimension_size": [
        6443,
        17999,
        36000
      ],
      "block_size": [
        5,
        1799,
        3600
      ],
      "attributes": {
        "comment": "\"Final\" version using Multi-Resolution Variational
 Analysis (MRVA) method for interpolation",
        "long_name": "analysed sea surface temperature",
        "standard_name": "sea_surface_foundation_temperature",
        "valid_max": 32767,
        "valid_min": -32767
      },
      "unit": "kelvin",
      "nodata_value": -32768,
      "offset": 298.149999999999977,
      "scale": 0.00100000000000000002
    },
    [ ... other variables, analysis_error, mask, sea_ice_fraction, omitted ...
 ]
  }
}
```

6

MULTIDIMENSIONAL COG IN PRACTICE

6 MULTIDIMENSIONAL COG IN PRACTICE

Terradue activity focuses on the validation of the GDAL GeoTIFF driver (D180) extended to implement the GDAL multidimension API in reading and writing.

6.1. Description

The use case backing this validation focuses on a data scientist, Alice, with a task for analyzing a times series of Earth observation data. Alice is tasked to analyze the impact of the wildfires that occurred in January 2020 over Kangaroo Island. Kangaroo Island lies 112 km (70 mi) southwest of Adelaide (Australia) and is regularly impacted by bushfires during the summer. Normally these bushfires are localized, but the summer of 2019-20 saw an unprecedented bushfire event that seriously impacted Kangaroo Island. The event started on the 20th December, 2019, from lightning strikes and when declared safe, on the 6th February, of the 440,500 hectare island, approximately 211,000 hectares were affected.

To analyze the impact of this bushfire event, Alice needs to obtain a temporal series of the Normalized Burn Ratio (NBR) over the area. The NBR is an index designed to highlight burnt areas in large fire zones. The formula is similar to NDVI, except that the formula combines the use of both near infrared (NIR) and shortwave infrared (SWIR) wavelengths.

The Copernicus Sentinel-2 mission satellites acquire 13 spectral bands in the visible and near-infrared (VNIR) and Short-wavelength infrared (SWIR) spectrum and provide the ideal input for this analysis.

The data is available as Cloud Processing Ready Data:

- The acquisitions are available as STAC items in a STAC catalog;
- The bands are exposed as assets in the COG format; and
- All resources are available via HTTP.

The use case will validate the GDAL GeoTIFF driver (D180) by generating a multidimensional COG by combining a stack of products, one per date with a vegetation index, and then consume the generated Multidimensional COG in a Jupyter Notebook.

6.2. Workflow

As illustrated in the figure below, the overall workflow that Alice needs to implement is to access the COG available data, calculate the indexes, and generate a multidimensional COG.

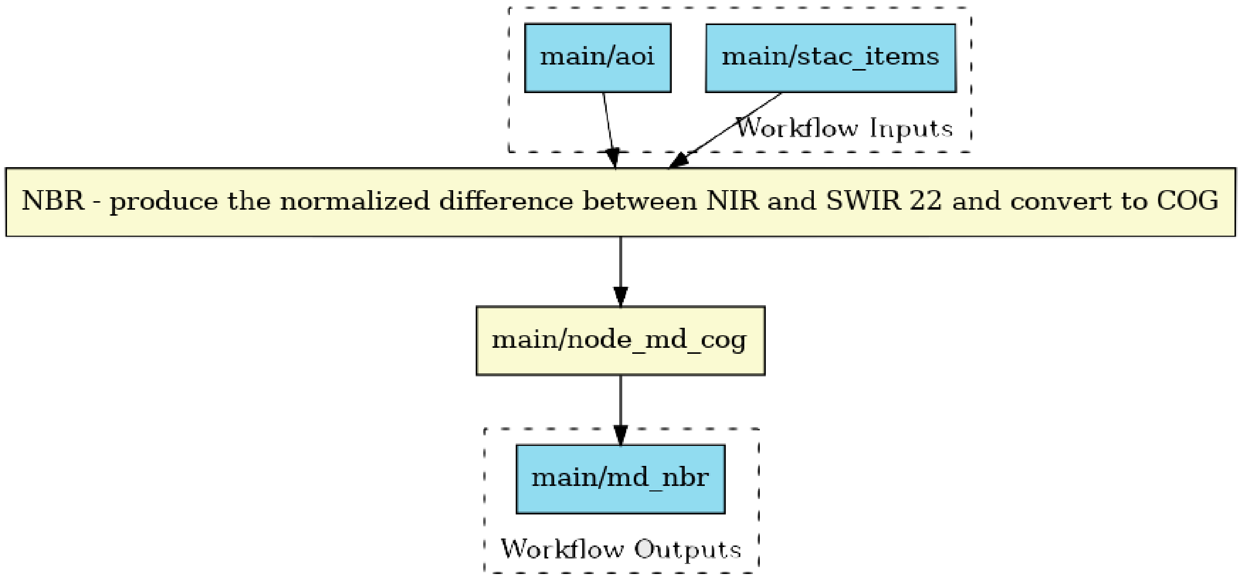

main/aoi
main/stac_items
Workflow Inputs
NBR - produce the normalized difference between NIR and SWIR 22 and convert to COG
main/node_md_cog
main/md_nbr
Workflow Outputs

Figure 2 — Overall data processing workflow

To achieve this, Alice will follow the Best Practice to package and deploy Earth Observation Applications in an Exploitation Platform (OGC 20-089). This Best Practice supports developers that want to adapt and package their existing algorithms written in a specific language to be reproducible, deployed, and executable in different platforms.

Alice will create Application Packages that describe the data processing applications, providing information about the parameters, software item, executable, dependencies and metadata. Alice builds a container image with her Application and command line tool(s) and respective runtime environments, publishes the container image on a repository, and writes the Application Package document with a workflow that invokes the command line tool(s) included in the image.

The Application Package is encoded in a Common Workflow Language (CWL) document to describe the Application, its parameters, command-line tools, their runtime environments, their arguments, and their invocation within containers. The CWL is a set of open standards for describing analysis workflows and tools in a way that makes them portable and scalable across a variety of software and hardware environments. The application package provides a well-defined set of procedures to allow “build to run” operations.

6.3. Implementation

The first application that Alice creates uses the new GDAL GeoTIFF driver (D180) to extract the study Area of Interest (AoI) and to create a Multidimensional COG. Alice writes a Common Workflow Language CWL document to:

- loop over a number of Sentinel-2 acquisitions to calculate the indexes; and

- stack the generated NBR in a Multidimensional COG.

As illustrated in the diagram below, for each Sentinel-2 the application will:

- clip over the area of interest;
- apply the normalized difference between the NIR and SWIR assets; and
- convert to COG.

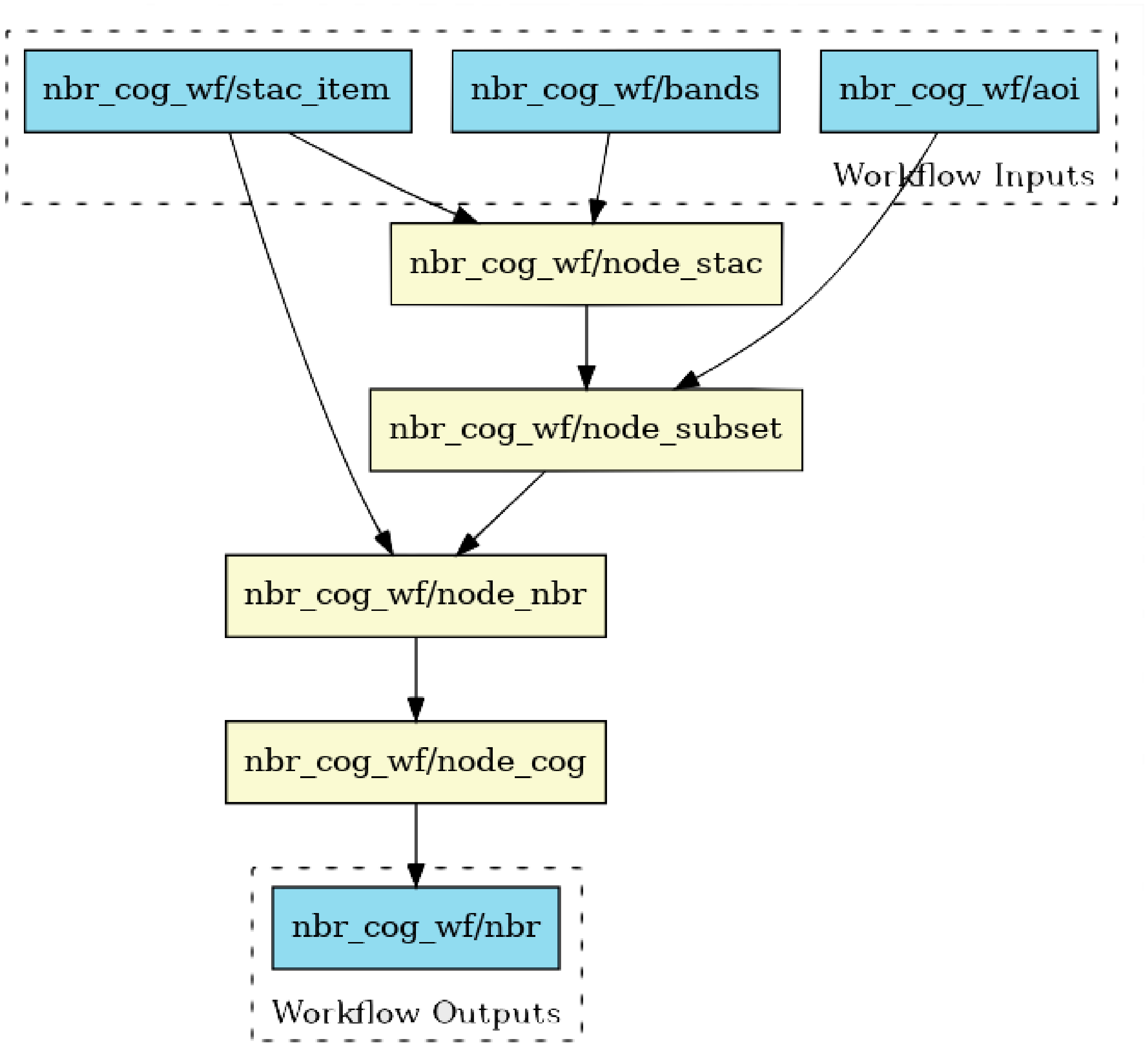


Figure 3 — Data processing flow for each product

The first step to clip the AoI is defined as a CWL CommandLineTool using GDAL_translate tool to extract the required area. As seen in the code below, the GDAL tool is available in a container directly provided by the OSGEO community.

```
- class: CommandLineTool
```

```
  id: translate_clt
  requirements:
    InlineJavascriptRequirement: {}
    DockerRequirement:
      dockerPull: docker.io/osgeo/gdal
  baseCommand: gdal_translate
  arguments:
  - -projwin
  - valueFrom: ${ return inputs.bbox.split(",")[0]; }
  - valueFrom: ${ return inputs.bbox.split(",")[3]; }
  - valueFrom: ${ return inputs.bbox.split(",")[2]; }
  - valueFrom: ${ return inputs.bbox.split(",")[1]; }
  - -projwin_srs
  - valueFrom: ${ return inputs.epsg; }
  - valueFrom: |
      ${ if (inputs.asset.startsWith("http")) {  return "/vsicurl/" + inputs.
asset; }
          else { return inputs.asset;}  }
  - valueFrom: ${ return inputs.asset.split("/").slice(-1)[0]; }
  inputs:
    asset:
      type: string
    bbox:
      type: string
    epsg:
      type: string
      default: "EPSG:4326"
  outputs:
    tifs:
      outputBinding:
        glob: '*.tif'
      type: File
```

The second important step is to calculate the required indexes. The application uses the Orfeo Toolbox (OTB), an open source software library for processing images from Earth observation satellites, and includes the argument to spectral indexes. The necessary application files are available in a specific container prepared by Terradue.

```
- class: CommandLineTool
  id: band_math_clt
  requirements:
    InlineJavascriptRequirement: {}
    DockerRequirement:
      dockerPull: docker.io/terradue/otb-7.2.0
  baseCommand: otbcli_BandMathX
  arguments:
  - -out
  - valueFrom: ${ return inputs.stac_item.split("/").slice(-1)[0] + ".tif"; }
  - -exp
  - '(im3b1 == 8 or im3b1 == 9 or im3b1 == 0 or im3b1 == 1 or im3b1 == 2 or
 im3b1 == 10 or im3b1 == 11) ? -2 : (im1b1 - im2b1) / (im1b1 + im2b1)'
  inputs:
    tifs:
      type: File[]
      inputBinding:
        position: 5
        prefix: -il
        separate: true
    stac_item:
      type: string
  outputs:
    nbr_tif:
      outputBinding:
        glob: "*.tif"
      type: File
```

The new GDAL GeoTIFF driver (D180) needs all inputs of the Multidimensional COG to be a COG file. For this we need to include an additional step in our workflow to transform the OTB output in a COG file. In the next step we use again the GDAL container available by OSGEO to make this transformation.

```
- class: CommandLineTool
  id: gdal_cog_clt
  requirements:
    InlineJavascriptRequirement: {}
    DockerRequirement:
      dockerPull: docker.io/osgeo/gdal
  baseCommand: gdal_translate
  arguments:
  - -co
  - COMPRESS=DEFLATE
  - -of
  - COG
  - valueFrom: ${ return inputs.tif }
  - valueFrom: ${ return inputs.tif.basename.replace(".tif", "") + '_cog.
tif'; }
  inputs:
    tif:
      type: File
  outputs:
    cog_tif:
      outputBinding:
        glob: '*_cog.tif'
      type: File
```

All the previous steps are managed by a CWL Workflow Class that assigns the respective inputs and outputs to create the vegetation index as described below. This workflow is repeated for each Sentinel-2 product and will be the inputs of the Multidimensional COG.

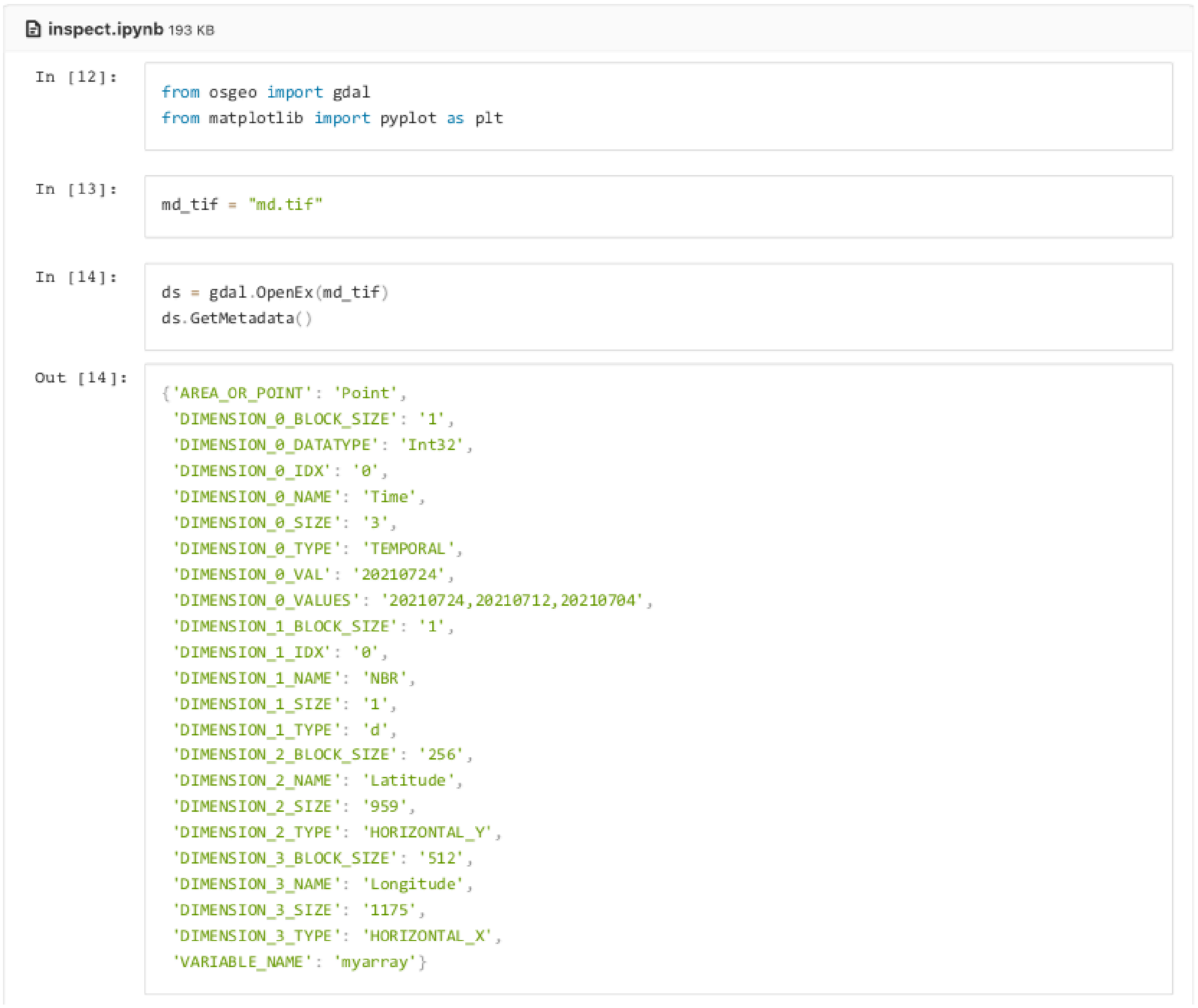
For the last step, Alice developed a Python application that uses the GDAL GeoTIFF driver (D180) to create the new Multidimensional COG with all the inputs previously created. As seen in the CWL below the application, including the new library, was included in a container, named md-cog, to ease its reproducibility.

```
- class: CommandLineTool
  id: md_cog_clt
  requirements:
    DockerRequirement:
      dockerPull: registry.gitlab.com/terradue-ogctb17/mcog
  baseCommand: mcog
  arguments:
  - --out
  - md.tif
  inputs:
    tif:
      type:
        type: array
        items: File
        inputBinding:
          prefix: -im
      inputBinding:
        position: 1
  outputs:
```

```
md_cog_tif:
  outputBinding:
    glob: 'md.tif'
  type: File
```

The full workflow validation process for the new GDAL GeoTIFF driver (D180) capability to write a Multidimensional COG was defined in an Application Package document, allowing the full experience to be reproducible, deployed, and executable in different platforms.

To validate the new GDAL GeoTIFF driver (D180) capability to read the Multidimensional COG, we implemented a Jupyter Notebook that analyses the metadata. The figures below show snippets of the notebook and the full code is available as an OGC resource.



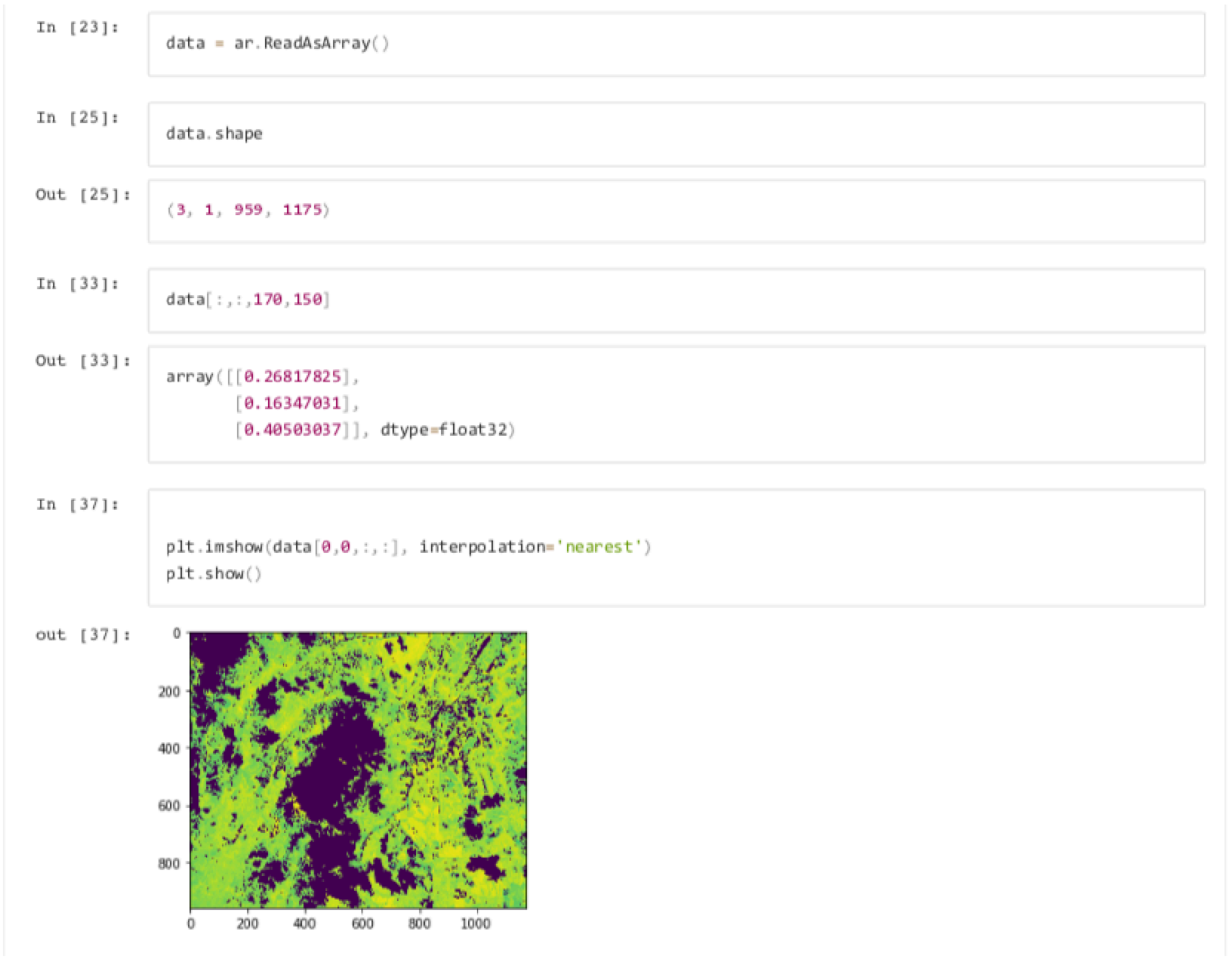
```
In [12]:
from osgeo import gdal
from matplotlib import pyplot as plt

In [13]:
md_tif = "md.tif"

In [14]:
ds = gdal.OpenEx(md_tif)
ds.GetMetadata()

Out [14]:
{'AREA_OR_POINT': 'Point',
 'DIMENSION_0_BLOCK_SIZE': '1',
 'DIMENSION_0_DATATYPE': 'Int32',
 'DIMENSION_0_IDX': '0',
 'DIMENSION_0_NAME': 'Time',
 'DIMENSION_0_SIZE': '3',
 'DIMENSION_0_TYPE': 'TEMPORAL',
 'DIMENSION_0_VAL': '20210724',
 'DIMENSION_0_VALUES': '20210724,20210712,20210704',
 'DIMENSION_1_BLOCK_SIZE': '1',
 'DIMENSION_1_IDX': '0',
 'DIMENSION_1_NAME': 'NBR',
 'DIMENSION_1_SIZE': '1',
 'DIMENSION_1_TYPE': 'd',
 'DIMENSION_2_BLOCK_SIZE': '256',
 'DIMENSION_2_NAME': 'Latitude',
 'DIMENSION_2_SIZE': '959',
 'DIMENSION_2_TYPE': 'HORIZONTAL_Y',
 'DIMENSION_3_BLOCK_SIZE': '512',
 'DIMENSION_3_NAME': 'Longitude',
 'DIMENSION_3_SIZE': '1175',
 'DIMENSION_3_TYPE': 'HORIZONTAL_X',
 'VARIABLE_NAME': 'myarray'}
```

Figure 4 — Notebook reading the Multidimensional COG metadata



```
In [23]:  data = ar.ReadAsArray()

In [25]:  data.shape

Out [25]: (3, 1, 959, 1175)

In [33]:  data[:,:,170,150]

Out [33]: array([[0.26817825],
                 [0.16347031],
                 [0.40503037]], dtype=float32)

In [37]:  plt.imshow(data[0,0,:,:], interpolation='nearest')
          plt.show()

out [37]:
```

Figure 5 — Notebook visualizing one specific dimension of the Multidimensional COG

6.4. Analysis

In the course of our activities, we were able to validate the new GDAL GeoTIFF driver (D180) capability to implement the GDAL multidimension API in reading and writing. We designed and implemented a reproducible use case that generates a multidimensional COG by combining a stack of products, one per date with a vegetation index, and then the generated Multidimensional COG is consumed in a Jupyter Notebook.

However, while the reading capability is demonstrably implemented to optimize the request of partial chunks spanning over several TIFFs, we observed two shortcomings on the writing capability of the new driver.

The first deals with the fact that the Area of Interest (AoI) must be aligned with the boundaries of the input's whole chunks. We understand that this is an implementation decision to avoid recompressing chunks that could have been previously partly written. This shortcoming is considered minor and easily circumventable in any implementation, but needs to be clearly documented to avoid user missteps.

The second shortcoming is more critical and has more concrete implications on the design of our user scenario. Initially our user scenario was focusing on a truly multidimensional dataset both in time and variable domains. Our goal was to create a multidimensional dataset including the satellite acquisition reflectances of several bands together with the calculated vegetation index across a temporal domain. Apart from the temporal analysis of a single index, our user would also have the possibility to analyze the actual values of any of the spectral bands available.

While theoretically possible by the design of the multidimensional COG format, the implementation provided by D180 is limited to one sample per array that led us to restrict the use case to a single vegetation index.

7

MULTIDIMENSIONAL COG IN JAVA

7 MULTIDIMENSIONAL COG IN JAVA

A COG reader was implemented in the open-source Apache Spatial Information System (SIS) project. This implementation can be an alternative to the widely-used GDAL library for Java developers. As a background, the next section below discusses some aspects to consider when binding a Java application to a native library such as GDAL. This section of the ER discusses the foundation offered by the standard Java platform for raster data. Then the latter part of this section summarizes the Apache SIS implementation and some observations based on this implementation experience are provided (e.g., on Clause 7.4.2 and Clause 7.4.3).

7.1. Note on bindings to native libraries

Given that GDAL is a widely used, extensively tested, open-source library supporting a large range of geospatial data formats, many projects want to be able to use it from a Java environment. However, while support for GDAL or other native library is often necessary, pure Java implementations for the most important formats are also valuable. At the time of writing this report, the only technology available in production for invoking C/C++ code from a Java program is the Java Native Interface (JNI). Those interfaces have been bundled in the Java Virtual Machine (JVM) since 1997. Some independent projects such as Java Native Access (JNA) or Java Native Runtime (JNR) provide easier ways to use a native library, but those projects either use JNI stubs or generate code at runtime for invoking JNI functions. Under the hood, the JNI remains the only access point.

The JNI requires that developers (or external frameworks such as JNR) write some C/C++ code as a “glue” between the Java environment and the native library. This glue may need to convert some data types. For example, an int is always 32 bits in Java, but of platform-dependent size in C/C++. Another purpose is to manage interactions with the Java garbage collector (GC). GC works in background threads and may move Java arrays to other memory locations at anytime. Non-Java code is not prepared to see pointers changing their values unexpectedly, possibly in the middle of a read or write operation. Consequently, when reading a GeoTIFF file with GDAL, giving GDAL a pointer where to write pixel values directly in the destination Java array is not provided. The usual workaround offered by JNI is to allocate a temporary array on the C/C++ heap, use that array for operations with the native library, then let JNI copy the C/C++ array to the Java array after the native library completed its operation. This approach can impact memory consumption and performance if the arrays are large, as often with raster data. JNI offers some mechanisms for avoiding copies (“get array critical” and “direct buffer”), but those alternatives are applicable only under limited circumstances.

Manually writing the C/C++ code required by JNI instead of relying on JNA or JNR can offer some advantages. This approach gives more control to the strategies for reducing array copies, and allows developers to combine many calls to the GDAL library for a single Java method call (i.e., developers are not forced to establish a one-to-one relationship between Java and native methods). The inconvenience is that writing JNI codes is time-consuming, requires knowledge in two languages (Java and C/C++), and forces developers to bundle native code for all supported platforms (in addition to the native library).

There is currently no ideal solution. The OpenJDK Panama project (still in incubation) is expected to provide the first JNI alternative fully supported by the Java Virtual Machine. Panama provides mechanisms for accessing "foreign memory" and "foreign functions" without the burden of writing JNI codes that create array copies. It was a D180 target of opportunity goal to test this binding with GDAL for reading GeoTIFF rasters. This goal will probably not be reached during the Testbed-17 due to the time frame. However, work on this experiment may continue afterward.

7.2. Why a pure Java implementation

Java applications are easier to distribute when there is no need to bundle platform-dependent native libraries. In addition, avoiding the cost of Java Native Interface (JNI) can improve performance. For example, mathematical functions rewritten in pure Java are reported faster than invoking equivalent code from the C/C++ FDLIBM library (JDK-8134780). Finally, a Java implementation can more easily be extended by another Java developer, using callback mechanisms (e.g., method overriding), which are difficult to apply when there is a language barrier between the library and the user. For example, Java developers can supply their own input streams fetching bytes from an unconventional source, such as a new cloud provider. C/C++ has an equivalent mechanism, but consuming in one language the input stream of another language can be difficult and inefficient.

7.3. Standard Java library and extensions

The standard Java2D library offers a rich framework for handling images. The foundations established in 1999 by Sun Microsystems and Eastman Kodak Company supports tiled images with various data types (integers from 1 to 32 bits and floating point values), pixel layouts (pixel interleaved, banded, packed) and color spaces (RGB, Cyan-Yellow-Magenta, *etc*). With this framework, it is possible for example to handle 1-bit (bilevel) images with no need to inflate all image data to a more common model such as 8 bits RGB. Conversions can be done on-the-fly on a pixel-by-pixel basis or for regions of interest.

The *Java Advanced Imaging* (JAI) extension from Sun Microsystems builds on this foundation for adding image processing capabilities with deferred execution, networked images (remote execution), chain of operations with a pull model, and more. While JAI does not seem to be maintained anymore, its design can be a source of inspiration providing all services needed for COG reader on top of Java2D.

A design goal of the COG reader is to expose its service through standard API as much as possible and to keep project-specific API to a minimum. This approach makes client code less dependent to this specific project and facilitates the use of COG data with other projects compliant with the same standards. Some APIs from the standard Java library used for handling COG images are:

- `java.awt.image.RenderedImage` for tiled image with deferred tile loading;
- `java.awt.image.SampleModel` for describing data type and pixel layout;
- `java.awt.image.ColorModel` for describing the color map; and
- `java.nio.channels.ByteChannel` for reading bytes from an arbitrary source (file, URL, cloud provider, *etc*).

Some APIs from OGC GeoAPI for handling COG geospatial metadata are:

- `org.opengis.referencing.crs.CoordinateReferenceSystem` (derived from ISO 19111) for data CRS; and
- `org.opengis.metadata.Metadata`, derived from ISO 19115, for title, author, geographic extent, *etc*.

Some APIs from the standard Java library not yet used, but planned:

- `java.nio.file.FileSystem` for virtual file system such as Amazon S3;
- `javax.imageio.ImageReader` for controlling the read operation (region of interest, subsampling, *etc*); and
- `javax.imageio.plugins.tiff.GeoTIFFTagSet` for constant values of GeoTIFF tags.

The standard Java library provides a TIFF reader and writer since Java 9. That reader/writer was ported from the Java Advanced Imaging project. It provides the values of GeoTIFF tags but without any processing. In particular it does not handle the *GeoKeys* referenced from `GeoKeyDirectoryTag`. Testing that reader for “raw” pixel values (without georeferencing) is a target of opportunity for this D180 work if time allows.

7.4. Apache SIS implementation

Apache Spatial Information System (SIS) is an open-source project focused on implementation of OGC standards. Implemented standards relevant to COG reader are ISO 19115 (metadata) and ISO 19111 (referencing by coordinates). In particular, in this testbed the version ISO 19115-1:2014 has been taken into account. OGC 19-008 (GeoTIFF) and OGC 09-083 (GeoAPI) are also important. Apache SIS provides its own implementation of a few data formats. The project supports fewer formats than GDAL, but implementation of a selected subset in an independent project offers the following advantages:

- Avoid native library (motivations discussed earlier);
- Leverage the rich metadata and CRS support of Apache SIS; and
- Share code between various formats (currently netCDF and GeoTIFF) implemented by the same library.

While metadata may be encoded in very different ways, there are some similarities in the way pixel values are encoded. In Apache SIS, the code reading, decompression, sub-setting, and organizing of pixel values in tiles is partially shared between netCDF and GeoTIFF readers. Improvements done for one format can benefit another format, and the amount of work needed for supporting new formats, is reduced. An interesting benefit for this Testbed is that a comparison of netCDF versus GeoTIFF performances would be less impacted by implementation differences because the two formats would have a larger percentage of common code (good or bad) compared to a comparison using the independent `netcdf` and `libtiff` C/C++ libraries. It was a target of opportunity goal to perform such comparison, but this work is postponed.

Because of shared foundation in Apache SIS, the GeoTIFF reader inherits from netCDF a capacity to handle *N*-dimensional data. The intend is to "plugin" the conventions defined by Spatialys in D180. This work is another target of opportunity postponed, but in theory it is only a matter of parsing the metadata defined by that convention.

7.4.1. Network accesses

Seek operations (moving to an arbitrary position in a stream of bytes) are efficient in memory, quite efficient on modern local file systems, but costly with remote connections on HTTP. To avoid this cost, the GeoTIFF reader tries to read the stream of bytes sequentially in a forward only direction. The TIFF format allows tiles and some TIFF tag values to appear at random positions in the file, but when reading an image region, Apache SIS plans its read operations in the "physical" order. This ordering should work also for the *n*-dimensional experiment mentioned in previous paragraph. It allows data producers to organize tiles in arbitrary order (not necessarily from left to right) and still read them without backward seek operations for a given geographic area. For example, we suspect that organizing tiles in Hilbert sequence order may offer performance benefits because this order tends to keep "physically close" tiles that are geographically close to each other, but we did not tested during this test bed.

While seek operations can be reduced, they can not be completely avoided. It is still necessary to skip a potentially large amount of bytes when the next tile to read is "physically" far. On servers supporting HTTP ranges, it may be more efficient to open a new connection with a list of ranges of bytes to be included in the responses. This is more complex than just moving to new positions: in order to avoid creating a new connection for each seek operation, the list of ranges should contain in advance all future seeks. The read order planning mentioned in previous paragraph helps to construct such HTTP ranges request. For verifying its effectiveness, a widget has been developed for visualizing the ranges of bytes read by Apache SIS (the widget can also be used with some other Java implementations).

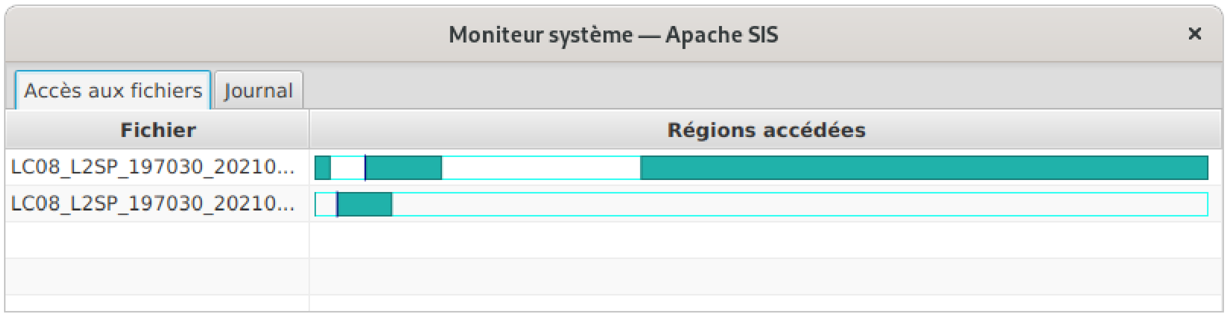

Moniteur système — Apache SIS
Accès aux fichiers
Journal
Fichier
Régions accédées
LC08_L2SP_197030_20210...
LC08_L2SP_197030_20210...

Figure 6 — Tracking the range of bytes read from two files

We have not yet measured the performance gain for images read from a distant server through HTTP.

7.4.2. BigTIFF

Big TIFF support was a design goal of Apache SIS GeoTIFF reader from the first day. By integrating early in the development phase the small difference between BigTIFF and regular TIFF, BigTIFF support requires very little effort. By contrast, modifying an existing reader afterward is probably more difficult.

7.4.3. ISO metadata

Metadata about the GeoTIFF file (title, author, geographic area, resolution, etc.) was represented according to the ISO 19115 metadata model. GeoAPI provides an "implementation" (from abstract standard point of view) of ISO 19115 as a set of software implementation neutral Java interfaces. Apache SIS provides an "implementation" (from Java language point of view) of GeoAPI metadata interfaces. The same metadata model it used for all raster formats supported by SIS, as well as vector formats, web services, aggregations, *etc.* The following table shows the mapping applied by Apache SIS from TIFF tags to ISO 19115 metadata elements.

Table 1

TIFF TAG	ISO METADATA
DateTime	identificationInfo/citation/date
ImageDescription	identificationInfo/citation/title
DocumentName	identificationInfo/citation/series/name
PageName	identificationInfo/citation/series/page
PageNumber	identificationInfo/citation/series/page

TIFF TAG	ISO METADATA
Artist	identificationInfo/citation/party/name
Copyright	identificationInfo/resourceConstraint
XResolution, YResolution	identificationInfo/spatialResolution/distance
“TIFF” (hard-coded)	identificationInfo/resourceFormat
Compression	identificationInfo/resourceFormat/fileDecompressionTechnique
BitsPerSample	contentInfo/attributeGroup/attribute/bitsPerValue
MinSampleValue	contentInfo/attributeGroup/attribute/minValue
MaxSampleValue	contentInfo/attributeGroup/attribute/maxValue
Threshholding	resourceLineage/processStep/description
HostComputer	resourceLineage/processStep/processingInformation/procedureDescription
Software	resourceLineage/processStep/processingInformation/softwareReference/title
Make	acquisitionInformation/instrument/citation/citedResponsibleParty/party/name (?)
Model	acquisitionInformation/platform/instrument/identifier

The following table lists ISO 19115 metadata elements inferred from GeoTIFF keys combined with TIFF tags.

Table 2

ISO METADATA
referenceSystemInfo
spatialRepresentationInfo/cellGeometry
spatialRepresentationInfo/pointInPixel
spatialRepresentationInfo/transformationDimensionDescription
spatialRepresentationInfo/transformationParameterAvailability

ISO METADATA
spatialRepresentationInfo/axisDimensionProperties/dimensionName
spatialRepresentationInfo/axisDimensionProperties/dimensionSize
spatialRepresentationInfo/axisDimensionProperties/resolution
identificationInfo/spatialRepresentationType
identificationInfo/extent/geographicElement

7.4.4. Testing framework

A `SelfConsistencyTest` class has been created for testing Apache SIS reader and other implementations. The test reads a whole GeoTIFF image at full resolution, then reads random sub-regions at random resolutions. The sub-regions pixels are compared with the original image. Assuming that the original (full resolution) image is correct, this test can detect some bugs in the code reading sub-regions or applying sub-sampling. This assumption is reasonable if we consider that the code reading the full image is often simpler than the code reading a subset of data.

Source code:

- GeoTIFF reader (ignoring dependencies)
- Self-consistency test class

8

COG AND ZARR: A COMMERCIAL USE CASE

8 COG AND ZARR: A COMMERCIAL USE CASE

Cities have grown and evolved for thousands of years, but today, more than ever, the velocity of urban development requires continuous monitoring to i) keep track continuously of the evolution of the territory, ii) assess the impact of decisions made, and iii) assess how a place — city, province, district, region or country — is performing according to Sustainable Development Goals (SDGs). In this context, there is a need to develop valuable tools to enhance the capacity of national, subnational, and local governments in data collection, mapping, analysis, and dissemination, building on a shared knowledge base using both globally comparable as well as locally generated data.

Latitudo 40 is working on developing an urban monitoring platform based on Earth Observation (EO) data. Below, the document refers to the platform with the name Urbalytics. The Urbalytics product simplifies access to information extracted through the processing of EO data, fully automating the entire life cycle of geospatial data, from the phase of search and acquisition of satellite images from the providers' repository (e.g., Copernicus, USGS, Airbus, Planet Labs, Capella Space, etc.) to their transfer in the platform data storage, to the application of processing algorithms. The services are accessed via a web dashboard or using REST APIs. It provides access to information on the status and trends of cities and regions to support urban and territorial development strategies and the local dimension of Sustainable Development Goals.

8.1. Urbalytics Use cases

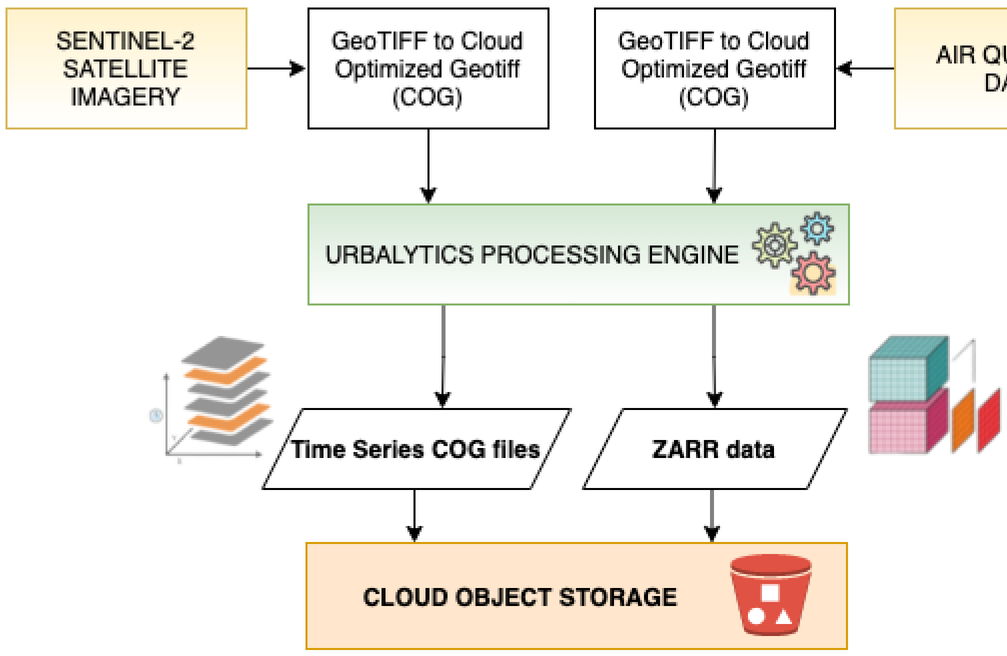

SENTINEL-2 SATELLITE IMAGERY
GeoTIFF to Cloud Optimized Geotiff (COG)
GeoTIFF to Cloud Optimized Geotiff (COG)
AIR QUALITY DATA
URBALYTICS PROCESSING ENGINE
Time Series COG files
ZARR data
CLOUD OBJECT STORAGE

Figure 7 — Diagram of Urbalytics test case

The size of the data to be managed, especially imagining a scalable scenario where the number of users who make requests for analysis grows rapidly, makes it necessary to find efficient methodologies to store and distribute spatial data for further use.

In the scope of this testbed, the following use cases were considered.

a) Time-series processing and analysis of satellite imagery for urban monitoring, e.g., search and acquisition of Sentinel-2 catalogues made available in Geotiff and COG format.

b) Time-series processing and analysis of data stored in multidimensional arrays (a.k.a. N-dimensional arrays, ND-arrays). For this testbed data from Copernicus Atmosphere Monitoring Service (CAMS) were taken into account. CAMS makes available regional air quality data in Europe, using an ensemble of 9 state-of-the-art numerical air quality models in NetCDF and GRIB format.

c) Creation, storage and distribution of derivative geospatial data on a cloud object storage.

8.2. Cloud Object Storage

Today, the storage and distribution of geospatial imagery face difficulties in cost-effectively storing an unprecedented amount of data. Since the arrival of Big Data, the most effective way to work with data is not to download it out of the cloud but rather to use distributed computing to process it from within the cloud to minimize data downloading. In the cloud, systems need to bring algorithms to the data, not the other way around. For this reason, the data storages have to be cost-effective, but at the same time enhance fast distributed processing. Cloud object storage is a data storage architecture for large stores of unstructured data with virtually infinite capacity and scalability. It is a service offered by all major cloud providers, e.g., AWS S3, Google Cloud Storage, Microsoft Azure Blob Storage. It designates each piece of data as an object, i.e., arbitrary collection of bytes, and bundles it with metadata and a unique identifier for easy access and retrieval. Data can be retrieved at any time using this identifier.

Object storage is inexpensive and cloud-friendly, but it arises some drawbacks to enable high-speed access for fast processing:

- access is via HTTP, i.e., not random disk reads; and
- use of HTTP range GET to retrieve a specific byte range.

In this context, COG and Zarr data format can fulfill the requirements with their approaches: i) organize data as an aggregation of small, independently retrievable objects (e.g., Zarr) and ii) allow access to pieces of large objects (e.g., Cloud-Optimized GeoTIFF).

So, for the reasons, the Urbalytics platform uses cloud object storage to store, process, and distribute data in COG and Zarr format.

8.3. KPIs evaluation

A study within the Urbalytics platform has been conducted to assess how COG and ZARR fulfill the requirements of the scenario depicted above. Following test cases were taken into account.

- Test case 1: Define an AoI overlapping two Sentinel-2 tiles. Sentinel-2 satellite images were available in COG format in a AWS S3 bucket, indexed by a STAC catalog. The AoI just overlaps 30% of both tiles. The test consisted of downloading all the dates available in the AWS collections for the specific AoI, then clip and process data to compute multispectral indexes. The next step was to execute the same test on both the collection with COG data and the collection with traditional GeoTiff files.
- Test case 2: Download Air Quality data in NetCDF format from CAMS, covering the whole of Italy from July 2018 to April 2021. Then, data were converted from NetCDF to ZARR data format offline and manually stored on an AWS S3 bucket. Finally, an algorithm for time-series forecasting was applied on both NetCDF and Zarr data, but considering just the city of Naples as Area of Interest.

In both test cases, an AoI covering only a portion of the data was chosen to assess the access by chunks that could boost the overall performance. The execution environment consisted of an AWS EC2 instance c5.2xlarge with 8 vCPU and 16GB of RAM. Tests were implemented in Python, using Xarray library.

For the assessment, the following quantitative criteria were chosen:

- **Execution time**: time to perform the whole processing (e.g., initial data download, multispectral index computation, time-series forecasting algorithm); and
- **Number of bytes transferred from/to the AWS S3 bucket**.

Benchmark results showed the following.

- A reduction by almost 54% on the total execution time for test case 1 and by almost 35% for test case 2, using respectively COG and Zarr format. A consistent reduction of the execution time was observed in the phase of data downloading. These results confirmed the advantage of the support for fine-grained access. Accessing a small area of interest within a file without downloading a large percentage of the overall data file consistently reduced the analysis time.
- In agreement with the previous point, a reduction of the number of bytes transferred from the cloud object storage was observed, almost by 30%. The reduction quota was slightly lower because both COG and ZARR data occupy more space than GeoTIFF and NetCDF data, respectively.

So, the experiments assessed a boost in the overall performance using the COG and ZARR data format. In addition, the transition in the Urbalytics platform from GeoTIFF and NetCDF to

COG and ZARR was easy, thanks to the excellent compatibility with the existing tools (GDAL, Jupyter / xarray, and QGIS) and first-class programming language support for Python.

8.4. Multimensional COG adoption

The GDAL GeoTIFF driver developed in this testbed enable GeoTIFF data to store multidimensional data. Adopting such a format could be beneficial in the context of the Urbalytics platform to store time-series of COG files in a single one. This could be beneficial regarding the number of files to store and manage in the system. Also, the distribution of a time-series of images could be arranged with one consistent file, limiting the number of files to pass, reducing error-prone activities during the exchange. However, some concerns arise about the final dimension of such a file that quickly goes beyond classic TIFF's 4 GB file size limit. Finally, above all, the actual implementation is also limited to a single multidimensional array per file. This limitation is highly restrictive for the Urbalytics use case, where multidimensional files with multiple variables were broadly used in the platform.

9

CONCLUSIONS AND FUTURE WORK

9 CONCLUSIONS AND FUTURE WORK

Geospatial data has traditionally been distributed via downloads from the data server to the local computer. This way of working suffers from limitations as datasets grow towards the petabyte scale. A cloud-based data repository offers several advantages over traditional data repositories — performance, reliability, cost-effectiveness, collaboration, and reproducibility. Despite the potential of cloud-native data repositories to accelerate scientific discovery, several challenges arise when traditional data formats have been applied to cloud-native environments.

The focus of this ER was documenting experiments in working with geospatial data in cloud-based environments, particularly with attention to the possibility of using two specific formats dedicated to managing the storage and distribution of images and data: COG and Zarr. Experiments conducted on commercial use cases showed clearly how COG and ZARR could improve the overall experience in a cloud environment. Support for fine-grained access enables access for a small area of interest within a file without downloading a large percentage of the overall data. When applied to data stored in cloud object storage, where the access is implemented by HTTP Get Range requests, this is a winning feature.

This Testbed attempted i) to evolve the Cloud Optimized GeoTIFF format to a multidimensional one and ii) to experiment with a cleaner solution for spatial referencing in Zarr format. During the validation of the Multidimensional GDAL GeoTIFF driver (D180), some concerns arose. In particular, the limitation of the actual implementation to a single multidimensional array per file seems to be very restrictive and needs further investigation. A typical user working on a multidimensional dataset should be able to analyze the values for each pixel both in time and for several variable domains. For this reason, testbed participants advise the continuous support for this activity and extension of the implementation to support multiple samples per array.

Finally, the following paragraphs list works that were “target of opportunities” for the Apache SIS implementation if time allowed, but which have been postponed to future versions.

9.1. Future works for Apache SIS implementation

9.1.1. Format comparisons

In format comparisons found on the internet, some points listed as advantage or inconvenient of a format are actually implementation characteristics of the most commonly used library for reading that format. For example the capability to store data as compressed chunks is often listed as a COG or ZARR advantage. But HDF5/netCDF-4 among others also has this capability (whether it has all the COG characteristics making it well suited for cloud environments has not been verified in this Testbed). Given two formats potentially well-suited to the cloud, whether the download traffic can be reduced with the other format as much as with COG depends a lot on the effort invested in the library implementation. For a more reliable comparison of

GeoTIFF, netCDF, and ZARR formats, we would need to do benchmarking over a larger range of implementations.

9.1.1.1. Native reader through Panama project

The traditional way to execute a task in an implementation-neutral way at OGC is with web services. But tests executed through a web services would actually benchmark the (reader + server) couple, making it more difficult to evaluate the characteristics of the reader alone. Tests using the API of a programmatic language would be more direct and reduce (but not eliminate) the intermediate layers.

If a test suite is written in Java, a binding to GDAL will be required. If a test suite is written in Python, a binding will still be needed for running the Java implementations. JNI/JNA/JNR bindings caveats have been mentioned previously. The Panama project alternative should be explored, either for using GDAL from Java or for using Java image readers from Python.

9.1.1.2. Leveraging netCDF reader for ZARR data

The ZARR format has some similarity with netCDF, in particular with their use of CF-conventions. The netCDF reader implementation in Apache SIS is already designed in a way where low-level operations are delegated to different codes (currently either UCAR library of SIS internal code). It should be possible to insert ZARR support in this framework. It would provide an additional reader implementation for the tests mentioned in previous paragraphs. Furthermore with ZARR and netCDF implementations sharing a significant amount of code, noise due to implementation differences would be reduced (but not eliminated) in format comparisons.

9.1.1.3. HDF5 support and comparison with other formats

The HDF5/netCDF-4 format was not on the list of tasks for this Testbed. But it may deserve a test using not only the library offered by the HDF group, but also at least one alternative implementation. In the context of Apache SIS it could be done in a way similar to ZARR, but with potentially more effort.

9.1.1.4. Hilbert sequence order of tiles

Storing tiles in a GeoTIFF file using Hilbert sequence order would keep close to each others tiles that are geographically close. It may reduce the amount of seek operations in some cases, which **may** have a positive effect on performance, but this is uncertain. Benchmarking could be done for testing this hypothesis.

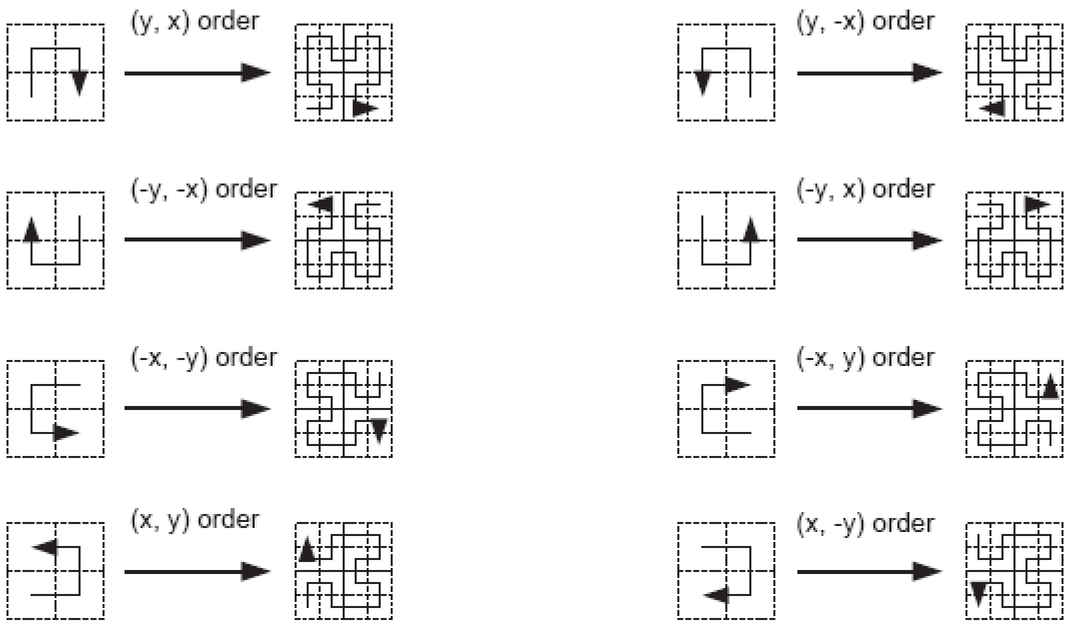


Figure 8 — Hilbert sequence order from ISO 19123:2007 (OGC 07-011) figure D.7

9.1.2. Spatialys conventions for Multi-dimensional GeoTIFF

Given that Apache SIS is designed for multi-dimensional coverages from the ground, it should be relatively easy to implement the multi-dimensional GeoTIFF proposal. Such experiment could be a source of feedback for the proposal.

A

ANNEX A (INFORMATIVE) REVISION HISTORY

A ANNEX A (INFORMATIVE) REVISION HISTORY

DATE	RELEASE	AUTHOR	PRIMARY CLAUSES MODIFIED	DESCRIPTION
May 17, 2021	.1	M. Manente	all	initial version
Oct 11, 2021	.2	M. Desruisseaux	all	Geomatys activities
Nov 11, 2021	.3	P. Goncalves	all	Terradue use case
Nov 16, 2021	.4	M. Desruisseaux	all	Network access and testing framework section
Nov 19, 2021	.5	M. Manente	all	final version

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Cloud Optimized GeoTIFF. An imagery format for cloud-native geospatial processing. In Internet: https://www.cogeo.org/, last accessed 2021/11/02

[2] GDAL COG Cloud Optimized GeoTIFF generator. In Internet: https://gdal.org/drivers/raster/cog.html last accessed 2021/11/02

[3] R. Fielding, Y. Lafon and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Range Requests (IETF RFC 7233). In Internet: https://datatracker.ietf.org/doc/html/rfc7233

[4] Joris Van Damme. The BigTIFF File Format. In Internet: https://www.awaresystems.be/imaging/tiff/bigtiff.html

[5] TIFF Revision 6.0 Final — June 3, 1992. Adobe Developers Association. In Internet: https://www.adobe.io/open/standards/TIFF.html

[6] Durbin, C., Quinn, P. and Shum, D., 2020. Task 51-Cloud-Optimized Format Study. In Internet: https://ntrs.nasa.gov/api/citations/20200001178/downloads/20200001178.pdf?attachment=true

[7] Richardson, M., Kearns, E. and O'Neil, J., 2020. Data dissemination best practices and challenges identified through NOAA's Big Data Project (No. EGU2020-12386). Copernicus Meetings.

[8] Hanson, M. and Leith, A., 2020, December. Sentinel-2 Cloud-Optimized GeoTIFF Public Dataset. In AGU Fall Meeting Abstracts (Vol. 2020, pp. IN042-0002).

[9] Roberts, N., Pieschke, R. and Lemig, K., 2019, December. Landsat in the cloud: Improving access and usability of the USGS Landsat record. In AGU Fall Meeting Abstracts (Vol. 2019, pp. IN22A-01).

[10] Abernathey, Ryan P., Joseph Hamman, and Alistair Miles. "Beyond netCDF: Cloud Native Climate Data with Zarr and XArray." AGU Fall Meeting Abstracts. Vol. 2018. 2018.

[11] Fisher, Ward, and Dennis Heimbigner. "NetCDF in the Cloud: modernizing storage options for the netCDF Data Model with Zarr." EGU General Assembly Conference Abstracts. 2020.

[12] Xu, Haiying, et al. "Using cloud-friendly data format in earth system models." AGU Fall Meeting Abstracts. Vol. 2019. 2019.

[13] Durbin, Chris, Patrick Quinn, and Dana Shum. "Task 51-Cloud-Optimized Format Study." (2020).

[15] Lu, Tianjian, et al. "Distributed Data Processing for Large-Scale Simulations on Cloud." 2021 IEEE International Joint EMC/SI/PI and EMC Europe Symposium. IEEE, 2021.

[16]Mozilla: HTTP headers — Range, https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Range