Cloud-optimized geo data formats and the Python ecosystem

Emilio Mayorga
Senior Oceanographer
Applied Physics Laboratory
University of Washington, Seattle
emiliom@uw.edu

CUGOS Spring Fling, 2023-04-21

Cloudy forecast

Nasty thunderstorms or pleasant clouds?

- Lots of large geospatial data out there, online, from many different providers. Much of it on "the cloud"
- But access alone doesn't equal convenient, scalable to data subsets
- Placing any old file on cloud storage doesn't guarantee scalable access

This presentation

- Focus on cloud-native data formats that support analytical use cases, not just visualization / cartographic use cases
- Primary geospatial types: **simple rasters**, **vector**, and **multi-dimensional grids**. I won't cover other types like point clouds (out of my wheelhouse).
- Birds eyeview, for the most part! I don't actively use some of this stuff
- Make things more concrete by presenting Python libraries that leverage these formats

Cloud storage, object storage

"Cloud storage" can mean different things. But in most cases, it refers to **"object storage"**, such as AWS "S3 buckets". All major cloud vendors provide object storage.

Object storage

- Unlike file systems, it's a "flat" storage. Key-value system, where the key is a unique identifier and the value is the data object. Can be made to mimic a hierarchical directory structure.
- Access via http protocol and access patterns
- "Infinitely" scalable: add more objects without impacting performance (unlike file systems)
- Encourages parallel processing to leverage this system characteristic
- Break up large files using a chunking (data partitioning) scheme that aligns with expected, most common access / query patterns.
- http byte range access or access to chunks as clearly organized and indexed files
- Flexible object metadata

Object storage, continued

- Promotes "file" (or object) based access, rather than relying on server software. APIs may facilitate access, but the access mechanism can operates on file objects that are cloud optimized. Simpler to deploy, simpler to build clients to.
- Can provide fast cloud-to-cloud access, especially when compute is from the same cloud provider and in the same cloud "region"
- Costs: Cloud access considerations
    - "Anonymous" (free, open) vs credentialled (free) vs "requester pays"
    - Access from outside the cloud vs within the same cloud provider but different region vs same region

`fsspec`: a unified Pythonic interface to local, remote, embedded file systems and bytes storage

- Access to files/objects on local file system, http, zip, S3 buckets, Azure blob storage, etc
- Wraps libraries from each cloud object storage provider in a common interface
- Provides uri-based access patterns: "s3://", "gcs://", "http://", "zip://", etc
- Provides common interface and returns Python "file-like objects" that can be chained by other Python libraries
- **https://filesystem-spec.readthedocs.io**

```
In [ ]: from pathlib import Path
        from aiobotocore.session import get_session
import fsspec

# 'ncei_wcsda' is a profile in my ~/.aws/credentials file
aws_profile_name = 'ncei_wcsda'

aws_session = get_session()
aws_session.set_config_variable('profile', aws_profile_name)
fs = fsspec.filesystem('s3', session=aws_session)

bucket = "ncei-wcsd-archive"
rawdirpath = "data/raw/Bell_M._Shimada/SH1707/EK60"

fs.ls(bucket)
```

```
In [ ]: ek60filedls = fs.ls(Path(bucket) / rawdirpath, detail=True)
        len(ek60filedls)
```

```
In [ ]: ek60filedls[:2]
```

Raster

Cloud-Optimized GeoTIFF (COG): **https://www.cogeo.org**. The first unmitigated success for a cloud-optimized geospatial data format.

- From **https://cogeo.org**: "COG relies on two complementary pieces of technology. The first is the ability of a GeoTIFF to not only store the raw pixels of the image, but to also organize those pixels in particular ways. The second is HTTP GET range requests, that let clients ask for just the portions of a file that they need. Together these enable fully online processing of data by COG-aware clients, as they can stream the right parts of the GeoTIFF as they need it, instead of having to download the whole file."
- Also fundamental: COG embeds in one file low-resolution overviews, tiles, and metadata accessible separate from the data.

- COGs leverage existing software that reads GeoTIFF. The software may tap into streaming capabilities, but can easily download the whole dataset and read it.
- **Python** software with COG suppport: `rasterio` (**https://rasterio.readthedocs.io**), `rioxarray` (**https://corteva.github.io/rioxarray/**), etc
- Serves both visualization and analysis use cases. RGB's for dynamic maps, though RGB's can also be used analytically, and the same format and tools are used for data access for analysis

Great, live demo: https://geotiffjs.github.io/cog-explorer/

Access COGs from `rasterio` via `fsspec`

Minimal examples of opening Cloud-optimized GeoTIFFs (COGs) with `rasterio`, where access is facilitated by `fsspec`. After opening, can define a geographical subset that will access only the overlapping COG tiles.

```
In [ ]: # ----- From http

import fsspec
import rasterio
import rasterio.plot as rioplot

uri = 'https://sentinel-cogs.s3.us-west-2.amazonaws.com/sentinel-s2-l2a-cogs/13/S/DV/2020/4/S2B_13SDV_20200428_0_L2A/TCI.ti

with fsspec.open(uri) as fobj:
    with rasterio.open(fobj) as riosrc:
        print(riosrc.profile)
```

In [ ]:
```
# ----- From an **S3 bucket** uri using an fsspec filesystem object

fs = fsspec.filesystem('s3', anon=True)
s3uri = 's3://sentinel-s1-rtc-indigo/tiles/RTC/1/IW/12/S/YJ/2016/S1B_20161121_12SYJ_ASC/Gamma0_VV.tif'

# Open and plot one of the overviews, not the full-resolution data
riosrc = rasterio.open(fs.open(s3uri), overview_level=3)
print(riosrc.profile)
rioplot.show(riosrc, cmap='gray', clim=(0, 0.4));
```

Access STAC Catalog that may contain collections of COGs

- STAC (SpatioTemporal Asset Catalog), **https://stacspec.org**
- pystac, **https://pystac.readthedocs.io**
- STAC catalogs can provide consistent access to multiple remote sensing granules

```
from pystac import Catalog

stac_catalog = 'https://drivendata-competition-building-segmentation.s3-us-west-1.amazonaws.com
/test/catalog.json'

test_cat = Catalog.from_file(stac_catalog)
```

Vector

- From **Chris Holmes, 2022**:
  - Vector is a lot more challenging to "cloud-optimize", compared to COGs
  - "For visualization it is really important to ‘prune’ most of the non-spatial attributes unless they are essential to rendering, or else the size of each tile will become quite large and thus slow to display. But for analysis it is important to be able to access all the attributes."
- Formats
  - **GeoJSON**: Web and Python-friendly but not realistic for medium-large data
  - **FlatGeobuf**: Efficient geometry encoding; fantastic streamable spatial index implementation. A domain format developed by the geospatial community.
  - **GeoParquet**: (**https://geoparquet.org**) A geospatial extension of the widely used, cloud-optimized "columnar" `Parquet` format for tabular data.

GeoParquet

- Still a new format with emerging take up
- BUT, it's based on Apache Parquet, which has huge takeup in tabular data applications on the cloud and a large ecosystem of tools that support it. Like COGs, GeoParquet benefits from an existing software ecosystem
- Columnar format, optimized for reading but writing is well supported. Widely used in data science applications.
- Chunked (partitioned) data, with chunks organized as files, and external metadata files
- **Python** support: `GeoPandas` (**https://geopandas.org**)
- QGIS supports it
- Great reads:
    - **https://getindata.com/blog/introducing-geoparquet-data-format/**
    - **https://kylebarron.dev/blog/geoparquet-on-the-web**

Great, live demo (but with `FlatGeoBuf`, **not** `GeoParquet`!): **https://flatgeobuf.org/examples/leaflet/large.html**

Access GeoParquet from `GeoPandas` via `fsspec`

- `(Geo)Pandas` implements `fsspec` internally, making for easier access patterns
- **https://geoparquet.org**
- **https://geopandas.org/en/stable/docs/reference/api/geopandas.read_parquet.html**

In [ ]:
```
# From an S3 uri using an fsspec filesystem object

from aiobotocore.session import get_session
import fsspec
import geopandas as gpd
import pandas as pd

# I'm cheating: This file is not actually a GeoParquet,
# though it does have lat-lon coordinates!
s3uri = 's3://trocas-data/rawresolution/gpsfiles_df.parq'

# Could pass filters to read_parquet, to request a data subset based on attribute values
df = pd.read_parquet(s3uri, storage_options={'profile': 'trocas'})
# gdf = gpd.read_parquet(s3uri, storage_options={'profile': 'trocas'})

print(f"{len(df)} records")
df.head()
```

```
In [ ]: gdf = gpd.GeoDataFrame(
            df,
    geometry=gpd.points_from_xy(df['longitude'], df['latitude'], crs="epsg:4326")
)

gdf.plot(markersize=2, column='TROCAS_nbr');
```

Multi-dimensional grids

- Grid data typically with >2 dimensions: (time,lat,lon), (band,lat,lon), (time,depth,lat,lon), etc
- Examples: Ocean model output. Rainfall gridded time series from remote sensing.
- Labelled arrays, with rich variable, coordinate and dataset metadata.
- Multi-dimensional chunks and compression.
- `netCDF` is the most widely used format. Recent developments allow for some cloud optimization based on internal chunking and HTTP Byte Range access, but that's not its strength. (though tools like `kerchunk` can now provide Zarr-like "virtual" cloud-optimized datasets out of netcdf collections. Similar to GDAL VRT's)
- `Zarr` (**https://zarr.dev**) is a cloud-optimized format that builds on the `netCDF` data model, conventions and software ecosystem.
    - Like Parquet, each chunk exists as a separate file object. Metadata stored separately in JSON files.
- **Python** support: `xarray` (**https://docs.xarray.dev**), `Zarr` (**https://zarr.readthedocs.io**), `rioxarray` (**https://corteva.github.io/rioxarray/stable/**). `xarray` is used extensively. Builds on and extends `Pandas` access patterns

```
In [ ]: # From https://pangeo-forge.org/dashboard/feedstock/78

import xarray as xr

zarr_store = 'https://ncsa.osn.xsede.org/Pangeo/pangeo-forge/pangeo-forge/AGDC-feedstock/AGCD.zarr'

ds = xr.open_dataset(zarr_store, engine='zarr', chunks={})

# Lazy loading (metadata and coordinate values only)
# Data variables loaded as lazy, chunked "Dask" arrays
print(f"Total size (not downloaded size!): {ds.nbytes/1e9:.1f} GB")
ds
```

In [ ]:
```
# This is slow; I'm not sure why.
ds['precip'].isel(time=-1).plot();
```

Various helpful links

- **https://schedule.cloudnativegeo.org/**
- **https://nasa-openscapes.github.io/earthdata-cloud-cookbook/** (but be aware of frequent cases where registration is required)
- **https://cholmes.medium.com/an-overview-of-cloud-native-vector-c223845638e0**
- **https://mapscaping.com/cloud-native-geospatial-vector-formats/**

```
In [ ]:
```