Evaluating Cloud-Optimized HDF5 for NASA's ICESat-2 Mission

Luis A. Lopez[1], Andrew P. Barrett[1], Amy Steiker[1], Aleksandar Jelenak[2], Lisa Kaser[1], Jeffrey E. Lee[3]

[1]CIRES, National Snow and Ice Data Center, University of Colorado, Boulder., Boulder, CO, USA
[2]The HDF Group, Champaign, IL, USA
[3]NASA Goddard Space Flight Center, NASA / KBR, Greenbelt, MD, USA

Abstract

The Hierarchical Data Format (HDF) is a common archival format for n-dimensional scientific data; it has been utilized to store valuable information from astrophysics to earth sciences and everything in between. As flexible and powerful as HDF can be, it comes with big tradeoffs when it's accessed from remote storage systems, mainly because the file format and the client I/O libraries were designed for local and supercomputing workflows. As scientific data and workflows migrate to the cloud , efficient access to data stored in HDF format is a key factor that will accelerate or slow down "science in the cloud" across all disciplines. We present an implementation of recently available features in the HDF5 stack that results in performant access to HDF from remote cloud storage. This performance is on par with modern cloud-native formats like Zarr but with the advantage of not having to reformat data or generate metadata sidecar files (DMR++, Kerchunk). Our benchmarks also show potential cost-savings for data producers if their data are processed using cloud-optimized strategies.

1 Problem

Scientific data from NASA and other agencies are increasingly being distributed from the commercial cloud. Cloud storage enables large-scale workflows and should reduce local storage costs. It also allows the use of scalable on-demand cloud computing resources by individual scientists and the broader scientific community. However, the majority of this scientific data is stored in a format that was not designed for the cloud: The Hierarchical Data format or HDF.

The most recent version of the Hierarchical Data Format is HDF5, a common archival format for n-dimensional scientific data; it has been utilized to store valuable information from astrophysics to earth sciences and everything in between. As flexible and powerful as HDF5 can be, it comes with big trade-offs when it's accessed from remote storage systems.

HDF5 is a complex file format; we can think of it as a file system using a tree-like structure with multiple data types and native data structures. Because of this complexity, the most reliable way of accessing data stored in this format is using the HDF5 C API. Regardless of access pattern, nearly all tools ultimately rely on the HDF5-C library and this brings a couple issues that affect the efficiency of accessing this format over the network:

1.0.1 Metadata fragmentation

When working with large datasets, especially those that include numerous variables and nested groups, the storage of file-level metadata can become a challenge. By default, metadata associated with each dataset is stored in chunks of 4 kilobytes (KB). This chunking mechanism was originally intended to optimize storage efficiency and access speed on disks with hardware resources available more than 20 years ago. In datasets with many variables and/or complex hierarchical structures, these 4KB chunks can lead to significant fragmentation.

Fragmentation occurs when this metadata is spread out across multiple non-contiguous chunks within the file. This results in inefficiencies when accessing or modifying data because compatible libraries need to read from multiple, scattered locations in the file. Over time, as the dataset grows and evolves, this fragmentation can compound, leading to degraded performance and increased storage overhead. In particular, operations that involve reading or writing metadata, such as opening a file, checking attributes, or modifying variables, can become slower and more resource-intensive.

1.0.2 Global API Lock

Because of the historical complexity of operations with the HDF5 format(The HDF Group, n.d.), there has been a necessity to make the library thread-safe and similarly to what happens in the Python language, the simplest mechanism to implement this is to have a global API lock. This global lock is not as big of an issue when we read data from local disk but it becomes a major bottleneck when we read data over the network because each read is sequential and latency in the cloud is exponentially bigger than local access (MDN, 2024) (Scott, 2020).

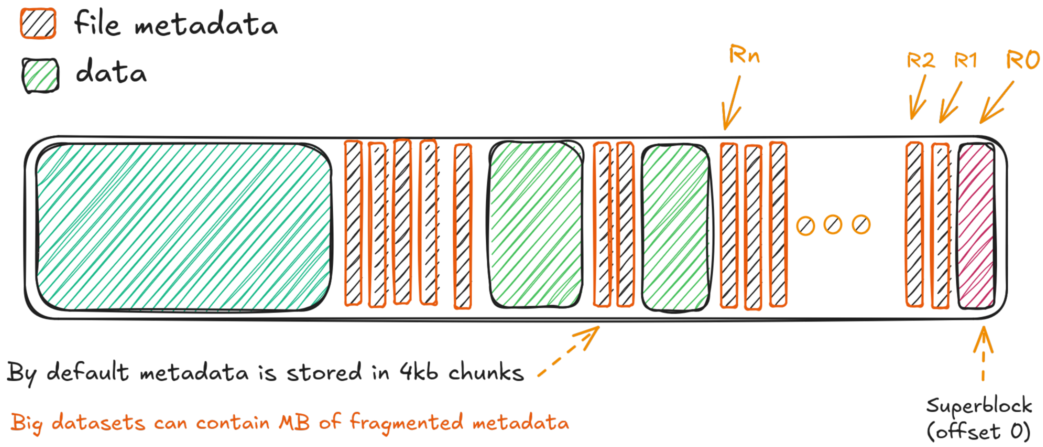

Figure 1: shows how reads (Rn) are done in order to access file metadata, In the first read, R0, the HDF5 library verifies the file signature from the superblock, subsequent reads, R1, R2,...Rn, read file metadata, 4kb at the time.

1.0.3 Background and data selection

As a result of community feedback and "hack weeks" organized by NSIDC and UW eScience Institute in 2023(ICESAT-2 HackWeek, 2023), NSIDC started the Cloud Optimized Format Investigation (COFI) project to improve access to HDF5 from the ICESat-2 mission, a spaceborne lidar that retrieves surface topography of the Earth's ice sheets, land and oceans (Neumann et al., 2019). Because of its complexity, large size and importance for cryospheric studies we targeted the ATL03 data product. The most relevant variable in ATL03 are geolocated photon heights from the ICESat-2 ATLAS instrument. Each ATL03 file contains 1003 geophysical variables in 6 data groups. Although our research was focused on this dataset, most of our findings are applicable to any dataset stored in HDF5 and NetCDF4.

2 Methodology

We tested access times to original and different configurations of cloud-optimized HDF5 ATL03 files stored in AWS S3 buckets in region us-west-2, the region hosting NASA's Earthdata Cloud archives. Files were accessed using Python tools commonly used by Earth scientists: h5py and Xarray(Hoyer & Hamman, 2017). h5py is a Python wrapper around the HDF5 C API. xarray[1] is a widely used Python package for working with n-dimensional data. We also tested access times using h5coro, a python package optimized for reading HDF5 files from S3 buckets and kerchunk, a

[1] `h5py` is a dependency of Xarray

tool that creates an efficient lookup table for file chunks to allow performant partial reads of files.

The test files were originally cloud optimized by "repacking" them, using a relatively new feature in the HDF5 C API called "paged aggregation". Page aggregation does 2 things: first, it collects file-level metadata from datasets and stores it on dedicated metadata blocks at the front of the file; second, it forces the library to write both data and metadata using these fixed-size pages. Aggregation allows client libraries to read file metadata with only a few requests using the page size as a fixed request size, overriding the 1 request per chunk behavior.

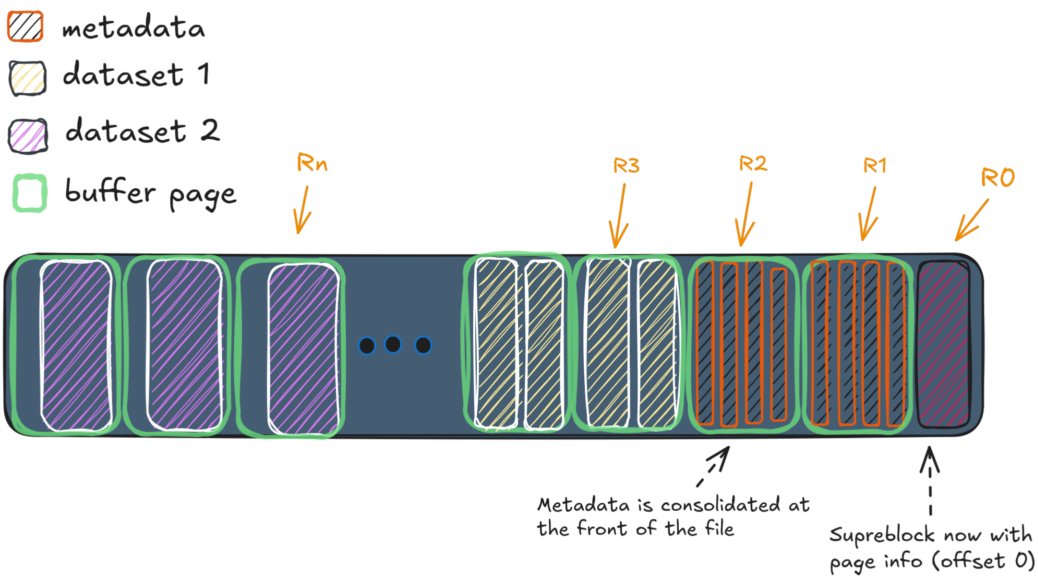

Figure 2: shows how file-level metadata and data gets internally packed once we use paged aggregation on a file.

As we can see in Figure 2, when we cloud optimize a file using paged-aggregation there are some considerations and behavior that we had to take into account. The first thing to observe is that page aggregation will – as we mentioned – consolidate the file-level metadata at the front of the file and will add information in the so-called superblock[2] The next thing to notice is page size is used across the board for metadata and data as of October 2024 and version 1.14 of the HDF5 library, page size cannot dynamically adjust to the total metadata size.

This one page size for all approach simplifies how the HDF5 API reads the file (if configured) but it also brings unused page space and chunk over-reads. In the case of the ICESat-2 dataset (ATL03) the data itself has been partitioned and each granule represents a segment in the satellite orbit and within the file the most relevant dataset is chunked using 10,000 items per chunk, with data being float-32 and using a fast compression value, the resulting chunk size is on average under 40KB, which is really small for an HTTP request, especially when we have to read them sequentially. Because of these considerations, we opted for testing different page sizes, and

[2] The HDF5 superblock is a crucial component of the HDF5 file format, acting as the starting point for accessing all data within the file. It stores important metadata such as the version of the file format, pointers to the root group, and addresses for locating different file components

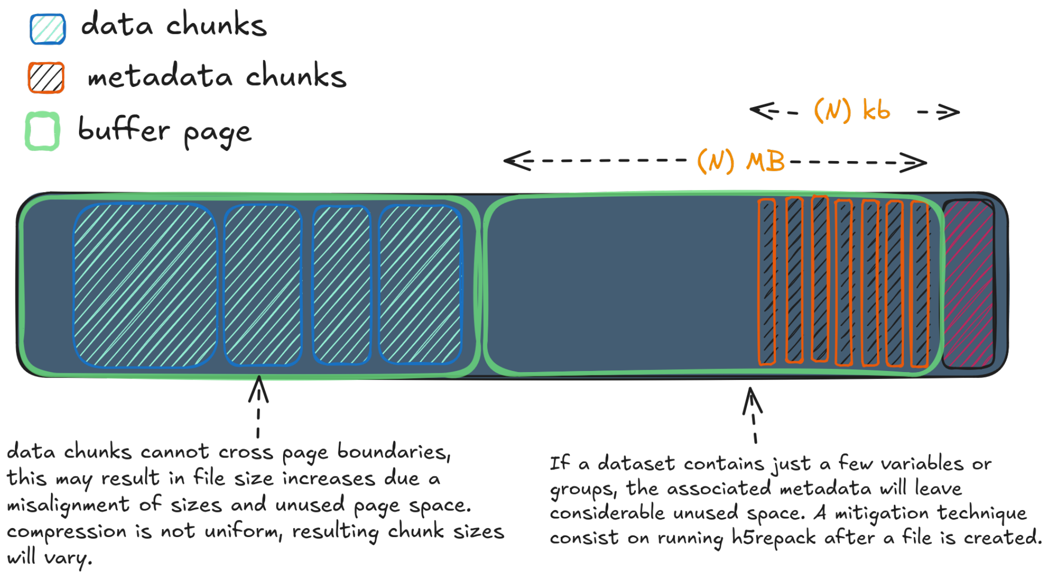

Figure 3: shows how file-level metadata and data packing inside aggregated pages leave unused space that can potentially increase the file size in a considerable way.

increase chunk size. The following table describes the different configurations used in our tests.

prefix	description	% file size increase	~km per chunk	shape	page size	avg_chunk_size
original	original file from ATL03 v6 (1gb and 7gb)	0	1.5km	(10000,)	N/A	35kb
original-kerchunk	kerchunk sidecar of the original file	N/A	1.5km	(10000,)	N/A	35kb
page-only-4mb	paged-aggregated file with 4mb per page	~1%	1.5km	(10000,)	4MB	35kb
page-only-8mb	paged-aggregated file with 4mb per pag8	~1%	1.5km	(10000,)	8MB	35kb
rechunked-4mb	page-aggregated and bigger chunk sizes	~1%	10km	(100000,)	4MB	400kb
rechunked-8mb	page-aggregated and bigger chunk sizes	~1%	10km	(100000,)	8MB	400kb
rechunked-8mb-kerchunk	kerchunk sidecar of the last paged-aggregated file	N/A	10km	(100000,)	8MB	400kb

This table represents the different configurations we used for our tests in 2 file sizes. It's worth noticing we encountered a few outlier cases where compression and chunk sizes led page aggregation to an increase in file size of approximately 10% which was above the desired value for NSIDC (5% max). We tested these files using the most common libraries to handle HDF5 and 2 different I/O drivers that support remote access to AWS S3, fsspec and the native S3. The results of our testing are explained in the next section and the code to reproduce the results is in the attached notebooks.

3 Results

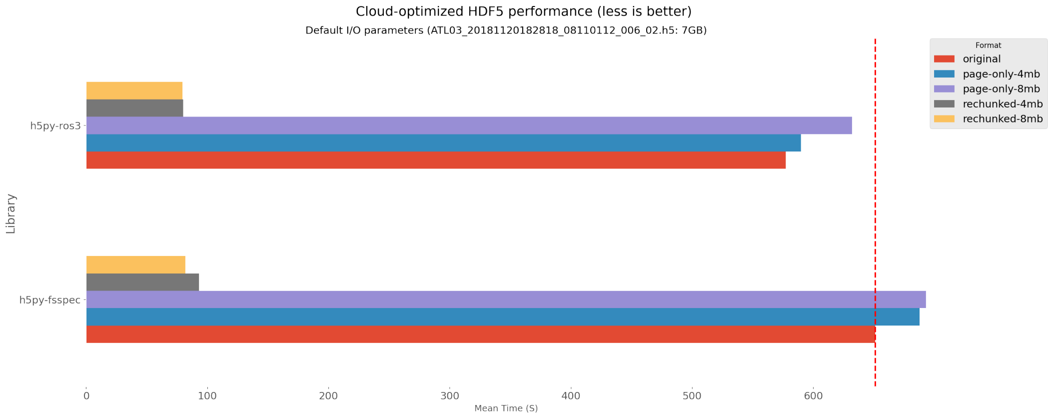

Figure 4: Using paged aggregation alone is not a complete solution. This behavior is caused by over-reads of data now distributed in pages and the internals of HDF5 not knowing how to optimize the requests. This means that if we cloud optimize alone and use the same code, in some cases we'll make access to these files even slower. A very important thing to notice here is that rechunking the file, in this case using 10X bigger chunks results in a predictable 10X improvement in access times without any cloud optimization involved. Having less chunks generates less metadata and bigger requests, in general is it recommended that chunk sizes should range between 1MB and 10MB[Add citation, S3 and HDF5] and if we have enough memory and bandwidth even bigger (Pangeo recommends up to 100MB chunks)[Add citation.]

4 Recommendations

Based on the benckmarks we got from our tests, we have split our recommendations for the ATL03 product into 3 main categories: creating the files, accessing the files, and future tool development. These recommendations aim to streamline HDF5 workflows in cloud environments, enhancing performance and reducing costs.

4.1 Recommended cloud optimizations

Based on our testing we recommend the following cloud optimizations for creating HDF5 files for the ATL03 product:

1. Create HDF5 files using paged aggregation by setting HDF5 library parameters:
   a. File page strategy: H5F_FSPACE_STRATEGY_PAGE
   b. File page size: 8000000 If repacking an existing file, h5repack contains the code to alter these variables inside the file

   ```
   h5repack -S PAGE -G 8000000 input.h5 output.h5
   ```

2. Avoid using unlimited dimensions when creating variables because the HDF5 API cannot support it inside buffered pages and representation of these variables is not supported by Kerchunk.

4.1.1 Reasoning

Based on the variable size of ATL03 it becomes really difficult to allocate a fixed metadata page. Big files contain north of 30MB of metadata, but the median metadata size per file is below 8MB. If we had adopted user block we would have caused an increase in the file size and storage cost of approximate 30% (reference to our tests). Another consequence of using a dedicated fixed page for file-level metadata

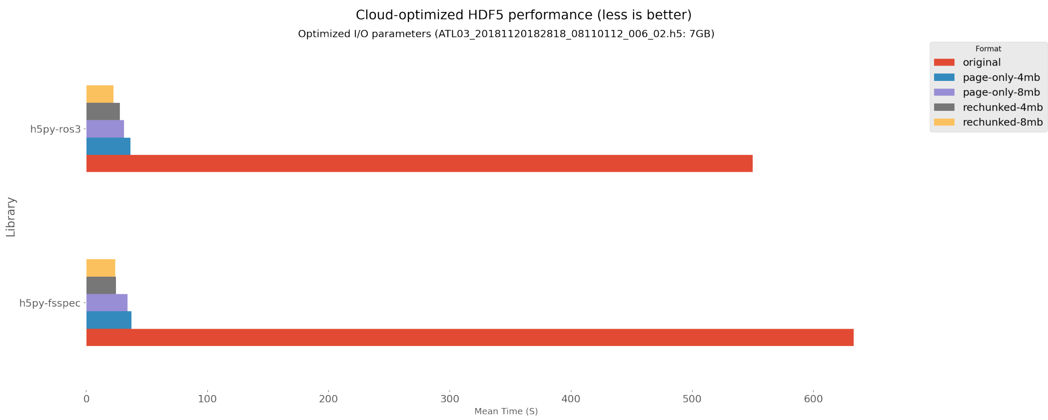

Figure 5: Once the I/O configuration is aligned with the chunking in the file, access times perform on par with cloud optimized access patterns like Kerchunk/Zarr. These numbers are from in-region execution. Out of region is considerably slower for the non-cloud-optimized case.

is that metadata overflow will generate the same impact in access times, the library will fetch the metadata in one go but the rest will be using the predefined block size of 4kb.

Paged aggregation is thus the simplest way of cloud optimizing an HDF5 file as the metadata will keep filling dedicated pages until all the file-level metadata is stored at the front of the file. Chunk sizes cannot be larger than the page size and when chunk sizes are smaller we need to take into account how these chunks will fit on a page, in an ideal scenario all the space will be filled but that is not the case and we will end up with unused space See 2.

4.2 Recommended Access Patterns

As we saw in our benchmarks, efficient access to cloud optimized HDF5 files in cloud storage requires that we also optimize our access patterns. The following recommendations focus on optimizing workflows for Python users. However, these recommendations should be applicable across programming languages. It's also worth mentioning that the HDF Group aims to include some of these features in their roadmap.

- **Efficient Reads**: Efficiently reading cloud-hosted HDF5 files involves minimizing network requests and prioritizing large sequential reads. Configure chunk sizes between 1–10 MB to match the block sizes used in cloud object storage systems, ensuring meaningful data retrieval in each read. Avoid small chunks, as they cause excessive HTTP overhead and slower access speeds.
- **Parallel Access**: Use parallel computing frameworks like `Dask` or multiprocessing to divide read operations across multiple processes or nodes. This alleviates the sequential access bottleneck caused by the HDF5 global lock, particularly in workflows accessing multiple datasets.
- **Cache Management**: Implement caching for metadata to avoid repetitive fetches. Tools like `fsspec` or `h5coro` allow in-memory or on-disk caching for frequently accessed data, reducing latency during high-frequency
- **Regional Access**: Operate workflows in the same cloud region as the data to minimize costs and latency. Cross-region data transfer is expensive and introduces significant delays. Where possible, deploy virtual machines close to the data storage region.

4.3 Recommended Tooling Development

To enable widespread and efficient use of HDF5 files in cloud environments, it is crucial to develop robust tools across all major programming languages. The HDF Group has expressed intentions to include these features in their roadmap, ensuring seamless compatibility with emerging cloud storage and computing standards. This section highlights tooling strategies to support metadata indexing, driver enhancements, and diagnostics, applicable to Python and other languages.

- **Enhanced HDF5 Drivers:** Improve drivers like `h5py` and `ROS3` to better handle cloud object storage's nuances, such as intelligent request batching and speculative reads. This mitigates inefficiencies caused by high-latency networks.
- **Metadata Indexing:** Develop tools for pre-indexing metadata, similar to Kerchunk. These tools should enable clients to retrieve only necessary data offsets, avoiding full metadata reads and improving access times.
- **Kerchunk-like Integration:** Extend Kerchunk to integrate seamlessly with analysis libraries like Xarray. This includes building robust sidecar files that efficiently map hierarchical datasets, enabling faster partial reads and enhancing cloud-native workflows.
- **Diagnostic Tools:** Create tools for diagnostics and performance profiling tailored to cloud-optimized HDF5 files. These tools should identify bottlenecks in access patterns and recommend adjustments in configurations or chunking strategies.

4.4 Mission implementation

ATL03 is a complex science data product containing both segmented (20 meters along-track) and large, variable-rate photon datasets. ATL03 is created using pipeline-style processing where the science data and NetCDF-style metadata are written by independent software packages. The following steps were employed to create cloud-optimized Release 007 ATL03 products, while minimizing increases in file size:

1. Set the "file space strategy" to H5F_FSPACE_STRATEGY_PAGE and enabled "free space tracking" within the HDF5 file creation property list.
2. Set the "file space page size" to 8MiB.
3. Change all "COMPACT" dataset storage types to "CONTIGUOUS".
4. Increase the "chunk size" of the photon-rate datasets (from 10,000 to 100,000 elements), while making sure no "chunk sizes" exceed the 8MiB "file space page size".
5. Introduce a new production step that executes the "h5repack" utility (with no options) to create a "defragmented" final product.

4.5 Discussion and Further Work

We believe that implementing cloud optimized HDF5 will greatly improve downstream workflows that will unlock science in the cloud. We also recognize that in order to get there, some key factors in the ecosystem need to be addressed. Chunking strategies, adaptive caching and automatic driver configurations should be developed to optimize performance.

Efforts should expand multi-language support, creating universal interfaces and libraries for broader adoption beyond Python. Cloud-native enhancements must focus on optimizing HDF5 for distributed systems and object storage, addressing egress costs, ease of use and scalability. Finally, advancing ecosystem interoperability involves setting integration standards and aligning with emerging trends such as serverless and edge computing. These efforts, combined with community collaboration, will modernize HDF5 to meet the challenges of evolving data-intensive applications.

4.5.1 Chunking Shapes and Sizes

Optimizing chunk shapes and sizes is essential for efficient HDF5 usage, especially in cloud environments:

- **Chunk Shape:** Align chunk dimensions with anticipated access patterns. For example, row-oriented queries benefit from row-aligned chunks.
- **Chunk Size:** Use chunk sizes between 1–10 MB to match cloud storage block sizes. Larger chunks improve sequential access but require more memory. Smaller chunks support granular reads but may increase network overhead.

Finally, we recognize that this study has not been as extensive as it could have been (cross language, multiple datasets) and yet we think we ran into the key scenarios data producers will face when they start producing cloud optimized HDf5 files. We think that there is room for improvement and experimentation with various configurations based on real-world scenarios is crucial to determine the best performance.

5 References

Hoyer, S., & Hamman, J. (2017). Xarray: N-D labeled arrays and datasets in python. *J. Open Res. Softw.*, *5*(1), 10.

ICESAT-2 HackWeek, H. C. (2023). h5cloud: Tools for cloud-based analysis of HDF5 data (Version v1.0.0). Retrieved from https://github.com/ICESAT-2HackWeek/h5cloud

MDN, M. (2024, May). Understanding latency. Retrieved from https://developer.mozilla.org/en-US/docs/Web/Performance/Understanding_latency

Neumann, T. A., Martino, A. J., Markus, T., Bae, S., Bock, M. R., Brenner, A. C., et al. (2019). The ice, cloud, and land elevation satellite – 2 mission: A global geolocated photon product derived from the advanced topographic laser altimeter system. *Remote Sensing of Environment*, *233*, 111325. https://doi.org/https://doi.org/10.1016/j.rse.2019.111325

Scott, C. (2020). Numbers every programmer should know. Retrieved from https://colin-scott.github.io/personal_website/research/interactive_latency.html

The HDF Group. (n.d.). Hierarchical Data Format, version 5. Retrieved from https://github.com/HDFGroup/hdf5