ESIP 2023: Accessing HDF5 data in the cloud with HSDS

John Readey
jreadey@hdfgroup.org

The HDF Group

Proprietary and Confidential. Copyright 2018, The HDF Group.

Overview

- Overview of HSDS
- Importing HDF5 files
- ICESat2 Case Study
- What's next
- Questions

HDF5 for the Cloud

- The HDF5 library has been around 20+ years, but is not optimized for cloud-native applications
  - Expects to access a filesystem (doesn't work well with object storage)
  - No remote API
  - No way to scale beyond one-thread/one-process (other than using MPI)
  - No support for multi-writer/multi-reader

Cloud Storage Models

- AWS S3 (or Azure Blob Storage, or Google Cloud Store) is natural choice for large data collections
  - Redundant
  - Can be accessed by multiple clients (compared with EBS Volumes)
  - Pay as you go pricing (pay for just what you use)
  - Low Cost – Compare 500 TB for one month:
    - S3 - $11K
    - EBS (EC2 attached volume, non-ssd) - $23K
    - EBS (EC2 attached volume, ssd) - $51K
    - EFS (sharable Posix compatible storage) - $41K

S3 Challenges for HDF5 data

- HDF5 Library either doesn’t work at all (for writing) or can be very slow (for reading)
  - Library was designed assuming POSIX compatibility & low latency
  - I/O operations for read can read small blocks of data at random locations
    - Latency Costs add up
    - Writing to S3 requires entire file to be updated (not practical for files of any size)
- Best throughput with S3 is with ~ 16 inflight requests with relatively large sizes
- These considerations led us to develop an entirely paradigm for persisting HDF5 data – the HDF object storage schema

HDF Cloud Native Schema

Why a sharded data format?

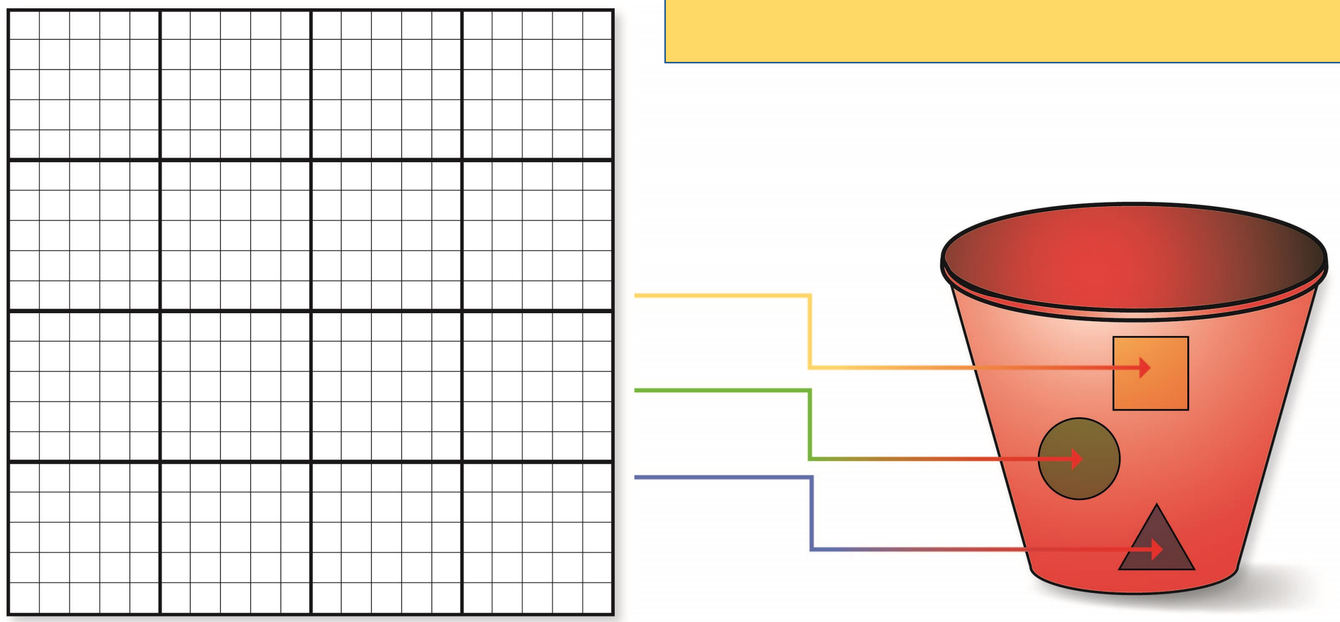
- Limit maximum size of any object
- Support parallelism for read/write
- Only data that is modified needs to be updated
- Multiple clients can be reading/updating the same “file”
- Don’t need to manage free space

Legend:

- *Dataset is partitioned into chunks*
- *Each chunk stored as an object (file)*
- *Dataset meta data (type, shape, attributes, etc.) stored in a separate object (as JSON text)*

Big Idea: Map individual HDF5 objects (datasets, groups, chunks) as Object Storage Objects

Each chunk (heavy outlines) get persisted as a separate object



Sharded format example


```
root_obj_id/
    group.json
    obj1_id/
        group.json
    obj2_id/
        dataset.json
        0_0
        0_1
    obj3_id/
        dataset.json
        0_0_2
        0_0_3
```

Observations:

- Metadata is stored as JSON
- Chunk data stored as binary blobs
- Self-explanatory
- One HDF5 file can translate to lots of objects
- Flat hierarchy – supports HDF5 multilinking
- Can limit maximum size of an object
- Can be used with Posix or object storage

Schema is documented here:
https://github.com/HDFGroup/hsds/blob/master/docs/design/obj_store_schema/obj_store_schema_v2.md

Introduction to HSDS

While it's possible to read/write the storage format directly, it's easier and more performant to use the HSDS (Highly Scalable Data Service). A REST based service for HDF.

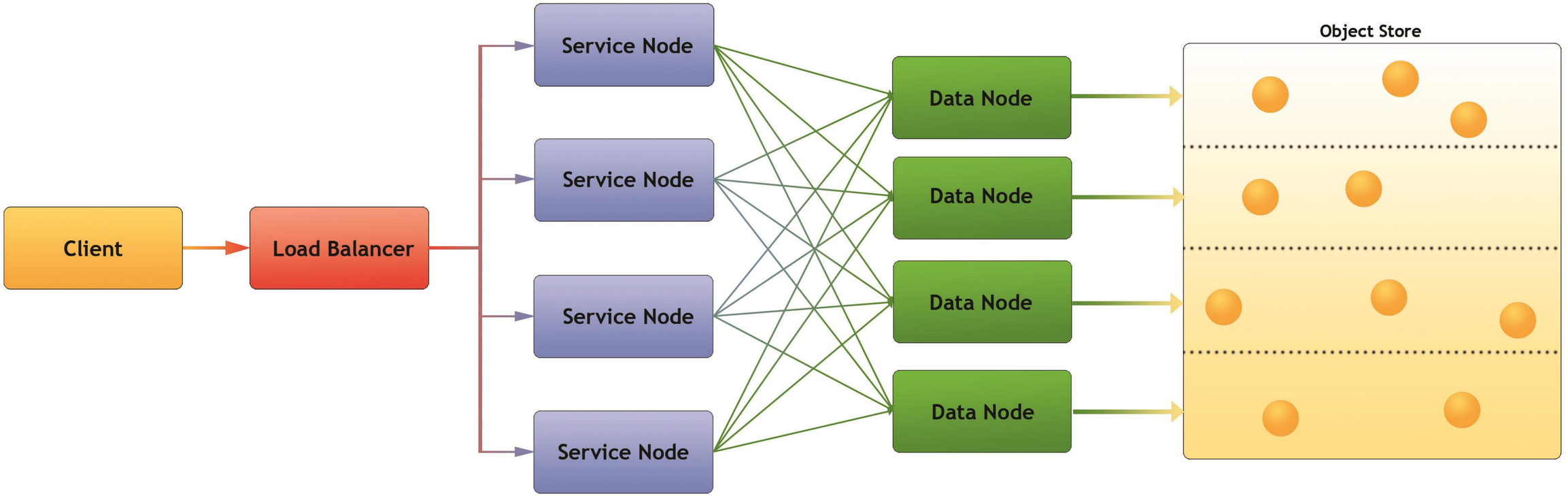
Note: in addition, there are serverless options for accessing the cloud native format, but we'll focus on HSDS for this talk

Software is available at:
https://github.com/HDFGroup/hsds

Server Features

- **Simple + familiar API**
  - Clients can interact with service using REST API
  - SDKs provide language specific interface (e.g. h5pyd for Python)
  - Can read/write just the data they need (as opposed to transferring entire files)
  - Support for compression
- **Container based**
  - Run in Docker or Kubernetes
- **Scalable performance**:
  - Can cache recently accessed data in RAM
  - Can parallelize requests across multiple nodes
  - More nodes ➔ better performance
  - Cluster based – any number of machines can be used to constitute the server
  - Multiple clients can read/write to same data source
  - No limit to the amount of data that can be stored by the service

Architecture



- Client: Any user of the service
- Load balancer – distributes requests to Service nodes
- Service Nodes – processes requests from clients (with help from Data Nodes)
- Data Nodes – responsible for partition of Object Store
- Object Store: Base storage service (e.g. AWS S3)

HSDS Platforms

HSDS can be run on most container management systems:

Using different supported storage systems:

POSIX
Filesystem

Accessing HSDS

- REST API
  - Language neutral
  - Support parallel and async
- H5pyd
  - Python 3.8 and up
  - Compatible with h5py
  - No parallel/async support
- HDF5 Library + REST VOL
  - C/C++
  - Parallel ops via H5DRead/Write multi call

Command Line Interface (CLI)

- Accessing HDF via a service means one can't utilize usual shell commands: ls, rm, chmod, etc.

- Command line tools are a set of simple apps to use instead:
  - hsinfo: display server version, connect info
  - hsls: list content of folder or file
  - hstouch: create folder or file
  - hsdel: delete a file
  - hsload: upload an HDF5 file
  - hsget: download content from server to an HDF5 file
  - hsacl: create/list/update ACLs (Access Control Lists)
  - Hsdiff: compare HDF5 file with sharded representation
- Implemented in Python & uses h5pyd
- Note: data is round-trip-able:
  - HDF5 File -> hsload -> HSDS store -> hsget -> (eqivalent) HDF5 file

Enabling HSDS access to HDF5 files

- Using hsload, HDF5 files can be transcoded to sharded schema
  - Will need roughly same amount of storage as original file(s)
  - Multiple files can be aggregated into one HSDS "file"
- To save time and extra storage costs, use "hsload –link". This will copy just the metadata and create pointers to each chunk in the source file. The storage needed for metadata is typically <1% of the overall file size
- To save even more time, use "hsload –fastlink". This will copy the meta data while chunk locations will be determined as needed. As chunk locations are found, they will be stored with metadata for faster access in future requests

Case Study – NASA ICESat-2 data

- ICESAT-2 mission collects precise (~4 mm resolution) elevation measurements as the satellite orbits over polar and non-polar regions
- Level 2 data is stored as HDF5 files stored on AWS S3
  - Roughly 3 GB and 1000 datasets per file
- As is often the case, the chunk size used (mostly 80 KB/chunk) is not optimal for cloud access
- Performance using ros3 VFD, s3fs, or HSDS linked files not very good
- Re-chunking PB's of HDF5 files is not very practical
- What to do?

Hyper-chunking

- In v 0.8.0 of HSDS, the "hyper-chunking" feature was added
- The idea of hyper-chunking is to use a larger chunk size than the source HDF5 dataset
- Each HSDS chunk is comprised of multiple adjacent HDF5 chunks in the HDF5 dataset
- Target size for HSDS chunk is 4-8 MB. In the case of ICESat-2 data, this would mean ~100 HDF5 chunks for each hyper-chunk
- This reduces the number of requests the DN nodes need to handle
- The DN nodes determine the set of actual S3 rangegets need to retrieve the HDF5 chunks

Intelligent Rangegets

- The store so far...
  - A DN node gets a request to read a HSDS hyper-chunk composed of multiple HDF5 chunks
  - It's often the case that the HDF5 chunks are adjacent (or nearly so) in the HDF5 file
  - The DN node will group nearby rangegets so one request can be made rather than two or more (the extra data has minimal impact on latency to S3)
  - In any case, requests to S3 are sent asynchronously which greatly improves performance

Benchmarking

- To compare relative performance for accessing ICESat-2 data on S3, we created a benchmark that replicates typical access patterns a science user would perform
- Python benchmark can be found here: https://github.com/HDFGroup/nasa_cloud/blob/main/benchmarks/python/icesat2_selection.py
- Based on the file path argument, data can be accessed as:
  - Local file on SSD
  - On S3 using ros3 VFD
  - On S3 using s3fs
  - Using HSDS (which reads from S3)

Results

- Local file: 0.7 s
- ROS3: 73.7 s
- S3FS: 36.7 s
- HSDS native: 19.3 s
- HSDS link (w/out hyper-chunking): 42.9 s
- HSDS link (w/ hyper-chunking): 37.8 s

Conclusions

- Performance of the ROS3 VFD not as good as S3FS
  - There are several opportunities for improvement though
- Data transcoded to the sharded format has more than a 2x performance advantage
  - But takes time and requires double the storage (if you are keeping the original files)
- Hyper-chunking performance was only modestly better than without hyper-chunking
  - Additional work is need. Would like to see link performance closer to sharded times

Acknowledgement

- This work was support by NASA contract: 80NSSC22K1744

Questions?