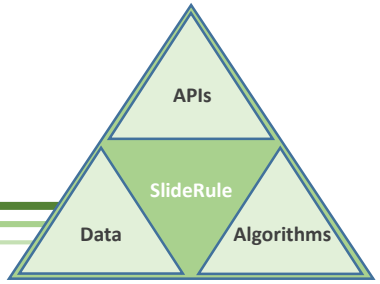


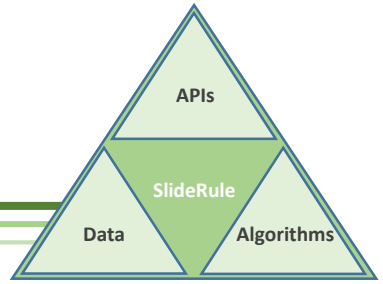
The Use of HSDS on SlideRule

HDF User's Group Meeting

JP Swinski/NASA/GSFC

October 13 - 16, 2020

Agenda

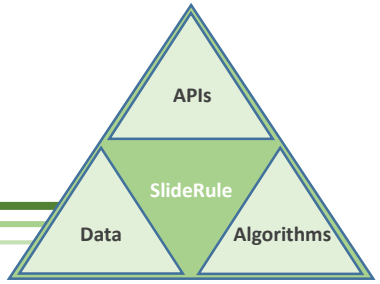


I. SlideRule Project Background

II. Why We Chose HSDS

III. Suggestions For Future HSDS Development

SlideRule Project Objectives



Project Objective: Promote new scientific discovery by lowering the barrier of entry to using the ICESat-2 data.

Tie-In to NASA's Mission: To make NASA datasets which are publicly available, practically accessible.

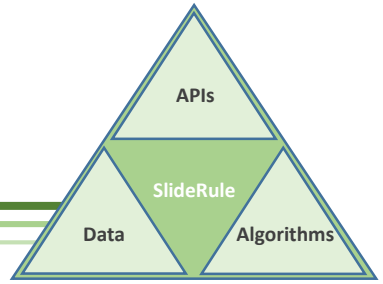
Problem Statement: The amount of data produced by ICESat-2 and the computational resources required by the algorithms that process the data, creates an often insurmountable barrier of entry to using the lower-level ICESat-2 data products. As a result, the typical use of ICESat-2 data is constrained to the pre-launch predicted science applications for which higher-level data products are generated.

Proposed Solution: Develop and deploy a publicly accessible ICESat-2 science data service that provides science data products generated on-demand using parameters supplied by researchers at the time of the request.

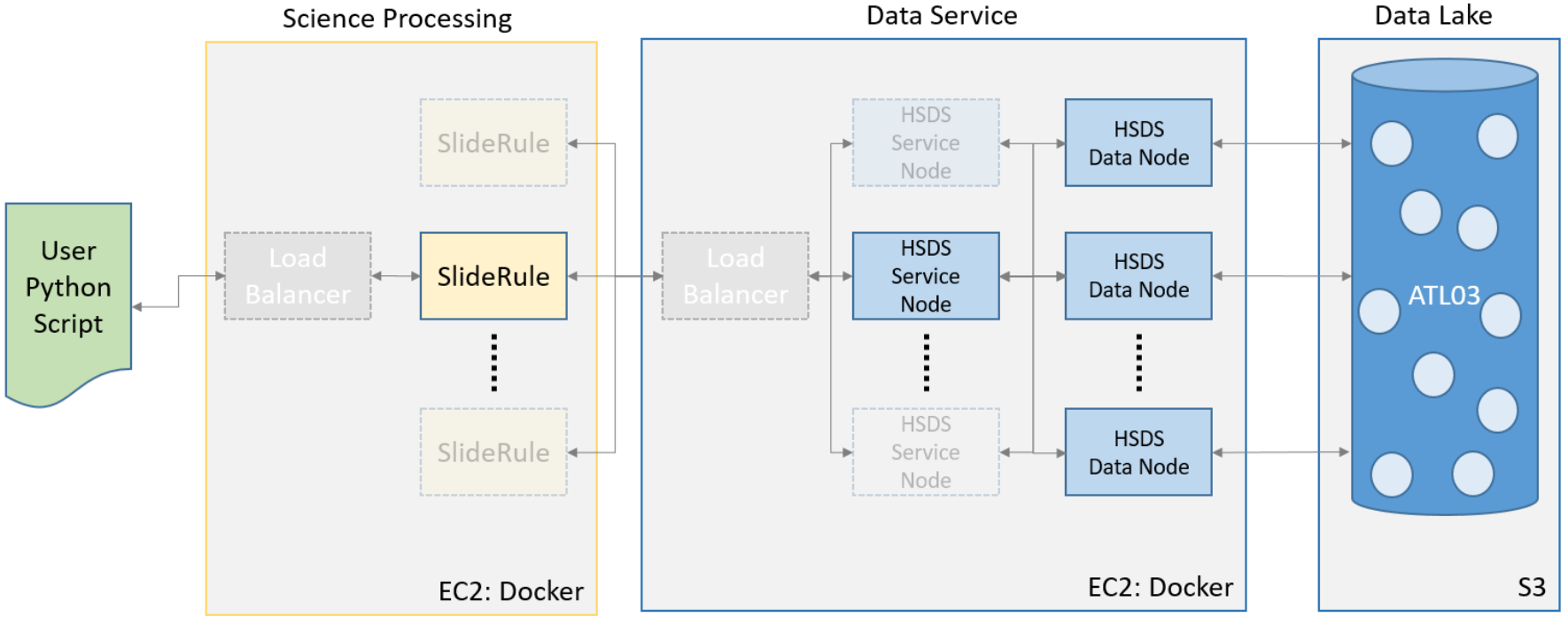
Key Benefits:

- There is a one-to-one mapping between resources spent producing data products and which data products are being used by the community.
- Unforeseen science applications are supported, with no additional cost, by the same system that supports the primary science objectives of the mission.
- The service-based architecture promotes integration with other agencies and organizations to improve the products and services they provide.
- Improvements to the algorithms that process the lower-level science data are immediately made available to the user communities (there is no longer a need to reprocess hundreds of terabytes of data, host the new version, and require users to re-download the data when a change is made in the processing algorithms).

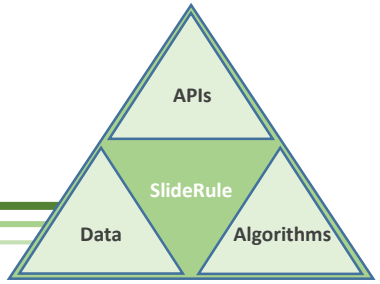
SlideRule Project Overview



- SlideRule is a server-side framework implemented in **C++/Lua** that provides **REST APIs** for processing science data and returning results.
- SlideRule communicates with a back-end data service implemented in **HSDS** with data stored in **AWS S3**.
- The initial target application for SlideRule is processing the lower-level **ICESat-2** point-cloud and atmospheric datasets for seasonal **snow depth** mapping and **glacier** research.



ICESat-2 Background Information



Mission

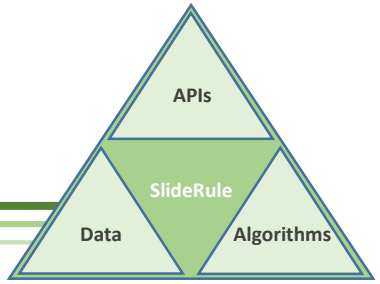
- Ice, Cloud, and land Elevation Satellite (ICESat-2) launched on September 15, 2018, with a three-year minimal mission life
- The Advanced Topographical Laser Altimetry System (ATLAS) is the sole instrument; it fires a laser towards earth 10,000 times a second and measures the amount of time it takes individual photons to reflect of the earth's surface and return back to the spacecraft.
- The individual photon time measurements are used to calculate surface elevations to a cm-level resolution.

Data

- ICESat-2 produces about 150 TB/year of low-level data.
- At 100Mbps egress, it would take 4.5 months to retrieve one year of data.
- Continuous aggregated egress rate varies depending on the network infrastructure. Over Internet2, rates as high as 400Mbps can be achieved, which still puts the time needed to retrieve the data at about five weeks.

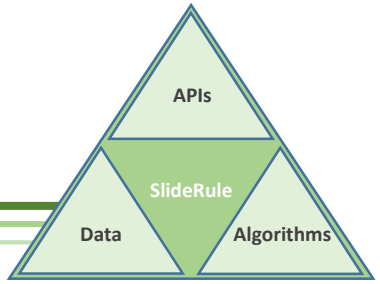
Algorithms

- ICESat-2 has two low-level data products – one used to study surface elevations, and one used to study atmospheric layers
- NASA provides seven Level-3A ICESat-2 data products covering a range of anticipated science applications
- NASA provides nine Level-3B ICESat-2 data products that target more specific science applications



Why We Chose HSDS

Solution #1



Move everything to the cloud and run it just like you would locally.

Drawback: It is higher cost for the same performance.

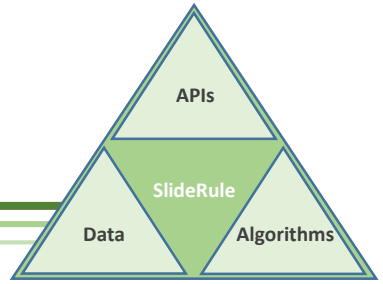
(1) Running cost effective cloud solutions require a different system architecture than what has been typically used for on premise compute and storage services

e.g. EFS storage of 150TB is ~$45K/month

(2) Typical data access patterns that are efficient for local file access, are not efficient when using object stores like S3.

e.g. when accessing a 2GB h5 file from a local disk, you pay an insignificant I/O penalty for data you don't read. But when using S3, it is not trivial to read just the data you need. You either pay for it in per-request latency overhead with multiple accesses to S3 to scan and traverse the file; or you pay for it all at once if you download the whole file to just read a little bit of it.

Solution #2

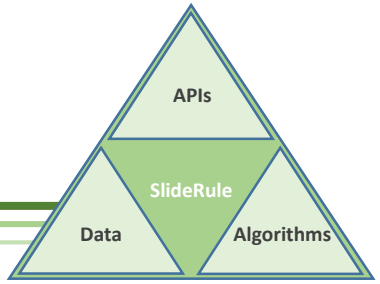


Transform the data into a cloud-optimized format.

Drawback: The ICESat-2 project has heavily invested in HDF5 based tooling, and the existing data pipelines took years to develop.

If we want to collaborate with and leverage off of the work other groups have done who have used the ICESat-2 datasets, then we must minimized the effort needed to adapt their tooling to our system.

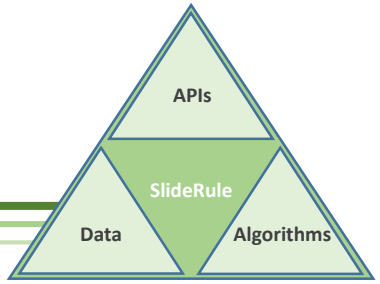
Solution #3



Run HSDS which maintains optimized indexes into the h5 files stored in S3 so that you can efficiently read only what you want.

Benefit: Existing tools can either be used as is or with minimal updates while still leveraging the benefits of a cloud-optimized architecture.

Solution #4



Run HSDS alongside an on-demand data processing engine (SlideRule).

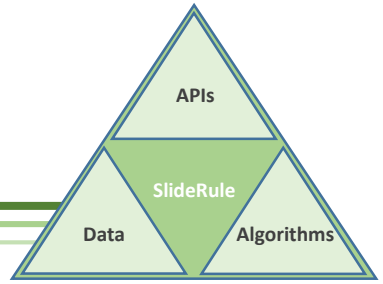
Benefit: Both cloud and client access.

(1) SlideRule provides a way to dynamically process the bulk of the data in the cloud using HSDS as a back-end, so that scientists can focus on the algorithms specific to their area of research.

(2) Power-users who have their own suite of lower-level algorithms can use the same HSDS services as a front-end and access the data directly (via h5pyd).

(3) Institutions can co-locate their processing in AWS (us-west-2) and leverage the full benefits of AWS infrastructure in processing the data.

Summary

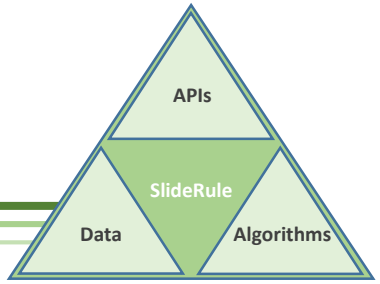


HSDS allows the use of existing datasets and tooling.

- HDF5 is ideally suited for archiving and provides a rich mechanism for metadata.
- There is a tremendous amount of NASA data currently in the HDF5 format.
- There is an extensive amount of tooling implemented by NASA and by external organizations that work with and expect the HDF5 format.
- The **h5pyd** package and **rest-vol** connector hide the physical cloud-access of the data from the logical access implemented in the code.

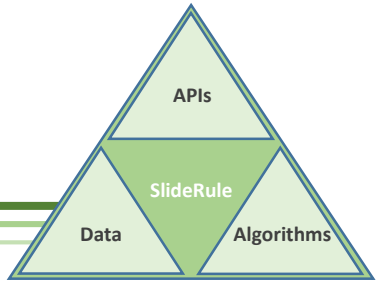
HSDS provides cloud-optimized access to existing datasets.

- Storing massive datasets in the cloud, cost effectively, for long periods of time, currently means storing those datasets in object stores (e.g. S3).
- The way data is read from an object store has different performance implications than the way it is read from a file system.
- HSDS provides performant access to HDF5 data in object stores, while providing the same logical view of the data as if it was stored locally.



Suggestions For Future HSDS Development

HSDS Link Option Performance

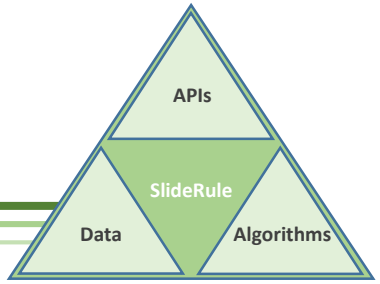


Decouple link option performance from dataset chunking

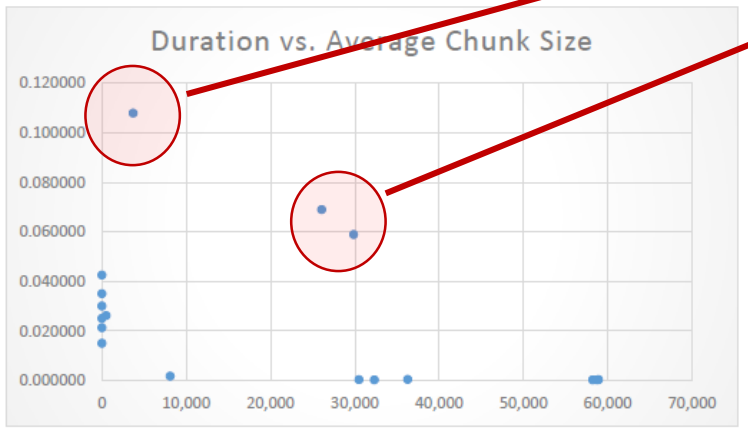
- When using the *native data ingest* mode for HSDS, each dataset is re-chunked to an optimal size for cloud access.
- The *native data ingest* mode requires the source dataset to be re-hosted and processed through an ETL pipeline. We want to leverage existing datasets in the cloud hosted by the institutions that own them.
- The *link option ingest* mode for HSDS allows the source data to remain where it is in S3, and indexes to the datasets inside the h5 files to be built and maintained by HSDS.

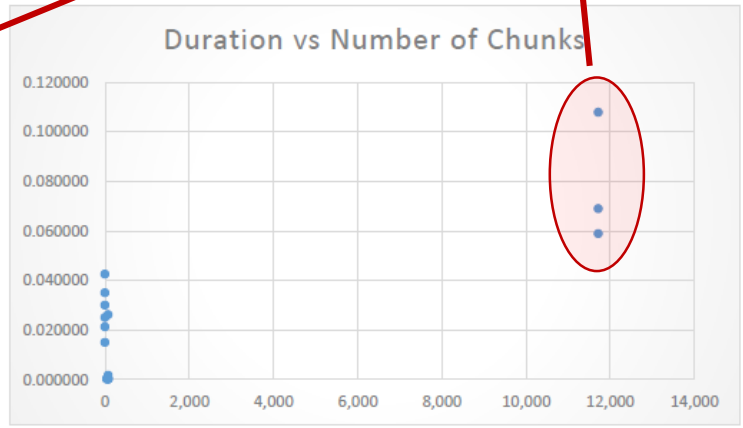
The problem is that the chunk sizes present in the source datasets may not be optimal for cloud access, and it turns out that **chunk size matters**.

HSDS Performance per Dataset



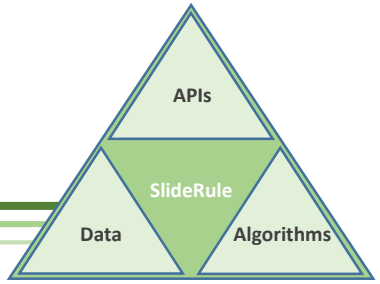
						Size	Elements	Chunks		Duration	
Dataset	Type	Storage Layout	Compression	Filters	Chunk Size	Total (bytes)	Total	Count	Average Size (bytes)	Average per Chunk (secs)	Average per Byte (secs)
/ancillary_data/atlas_sdp_gps_epoch	64-bit float	Compact	None	None	1	8	1	1	8	0.120	0.015000
/orbit_info/sc_orient	8-bit integer	Chunked	None	None	16	16	1	1	16	0.340	0.021250
/ancillary_data/start_rgt	32-bit integer	Compact	None	None	1	4	1	1	4	0.120	0.030000
/ancillary_data/end_rgt	32-bit integer	Compact	None	None	1	4	1	1	4	0.100	0.025000
/ancillary_data/start_cycle	32-bit integer	Compact	None	None	1	4	1	1	4	0.170	0.042500
/ancillary_data/end_cycle	32-bit integer	Compact	None	None	1	4	1	1	4	0.140	0.035000
<spot>/geolocation/delta_time	64-bit float	Chunked	GZIP: 6	None	10000	2,685,617	730,524	74	36,292	0.032	0.000396
<spot>/geolocation/segment_ph_cnt	32-bit integer	Chunked	GZIP: 6	Shuffle: 4	10000	600,951	730,524	74	8,121	0.032	0.001733
<spot>/geolocation/segment_id	32-bit integer	Chunked	GZIP: 6	Shuffle: 4	10000	38,586	730,524	74	521	0.031	0.026178
<spot>/geolocation/segment_dist_x	64-bit float	Chunked	GZIP: 6	None	10000	4,361,058	730,524	74	58,933	0.035	0.000267
<spot>/geolocation/reference_photon_lat	64-bit float	Chunked	GZIP: 6	None	10000	4,310,116	730,524	74	58,245	0.030	0.000230
<spot>/geolocation/reference_photon_lon	64-bit float	Chunked	GZIP: 6	None	10000	4,351,244	730,524	74	58,801	0.029	0.000222
<spot>/heights/dist_ph_along	32-bit integer	Chunked	GZIP: 6	None	10000	350,051,255	117,212,160	11,722	29,863	0.025	0.058842
<spot>/heights/h_ph	32-bit integer	Chunked	GZIP: 6	None	10000	305,855,771	117,212,160	11,722	26,092	0.026	0.068913
<spot>/heights/signal_conf_ph	8-bit integer	Chunked	GZIP: 6	Shuffle: 1	10000	43,666,117	117,212,160	11,722	3,725	0.006	0.107815
<spot>/bckgrd_atlas/delta_time	64-bit float	Chunked	GZIP: 6	None	10000	1,281,096	411,830	42	30,502	0.032	0.000261
<spot>/bckgrd_atlas/bckgrd_rate	32-bit integer	Chunked	GZIP: 6	None	10000	1,358,123	411,830	42	32,336	0.029	0.000226
						718,559,974	356,843,290	35,700	20,204	0.076	0.433833





Performance is dominated by the number of chunks being read. This suggests that the per-chunk overhead drives performance.

HSDS Performance per Data Format

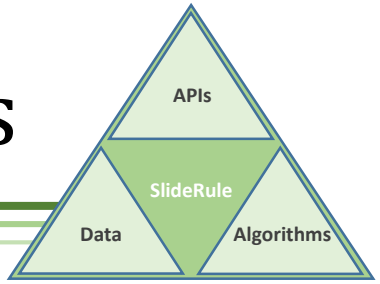


Format	Instance Type	Data Nodes	Http Compression	Duration (seconds)
Local File	c5.xlarge	n/a	Yes	10
Native Ingest	c5.xlarge	4	Yes	60
Link Option	c5.xlarge	4	Yes	990 to 1050
Link Option	c5.xlarge	4	**No**	1150
Link Option	c5.xlarge	**8**	Yes	990
Link Option	**c5.4xlarge**	**16**	Yes	620
Link Option w/ continuous data	c5.xlarge	4	Yes	30

Notes:

1. A total of 718MB of data was read out of 72 different datasets inside a large (~2GB) h5 file.
2. The c5.xlarge instance has 4 cores, 8GB of RAM, and up to 10Gbps of network connectivity.
3. All test runs used only one service node.

When Using HDF5, Chunking *Sometimes* Matters



Chunking a dataset can make a difference when:

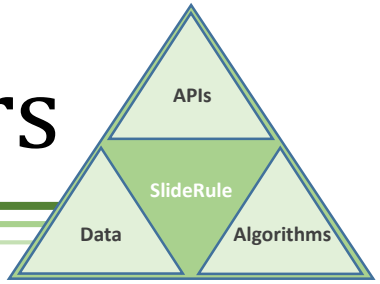
- **The datasets cannot fit in memory** – e.g. datasets that are larger than 8GB; individual chunks can be loaded into memory and worked on, and then swapped out for the next chunk.
- **The chunks can be identified a priori**, and only the necessary chunks are read from the dataset – e.g. bands of a multispectral image.

Chunking a dataset may not make a difference when:

- **The entire dataset is always read** – e.g. the lowest level of differentiation within an h5 file turns out to be the dataset itself
- **The chunks do not correlate to how the dataset is read** by end-users – e.g. multiple chunks need to be read in order to process the data; how many chunks need to be read before reading the entire dataset becomes just as efficient?

On a local file system, the overhead associated with reading chunks is negligible, and so creating smaller chunks improves latency with minimal cost to throughput.

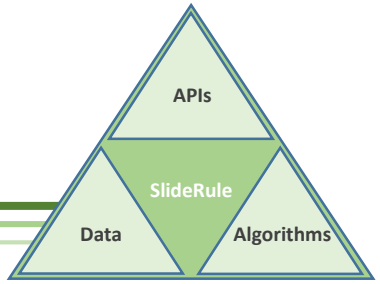
When Using HSDS, Chunking *Always* Matters



On a distributed system (e.g. object store), the overhead associated with reading chunks dominates the performance. Therefore both throughput and latency can be drastically affected by the wrong chunk size.

- Too large and small requests pay the price of reading much more data than is needed.
- Too small and large requests pay the price of expensive per-object S3 retrieval overhead.

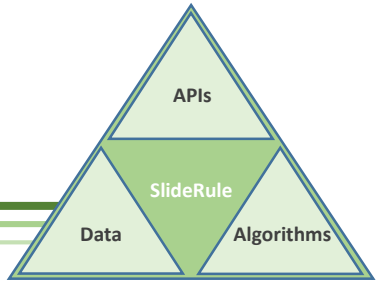
Continued Work



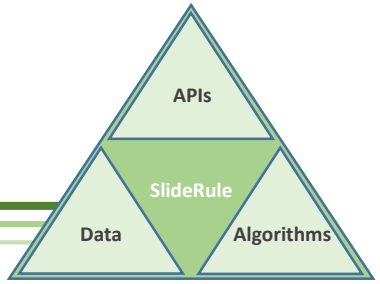
- Where are the bottlenecks?
  - S3 object – with native, the chunks are spread out across many objects, but with link, all the chunks ultimately point back to the same object which is stored at the same physical place in S3.
  - HSDS server – which parameters maximize CPU use and memory such that the network interface is saturated.
  - Client – even if the data throughput is very high, if the client doesn't need all of the data, then the effective data rate can be drastically reduced (e.g. a large chunk size may return more data to the client than needed such that to get the N bytes it wants, it has to receive x*N bytes).

- What does the performance curve look like across these different dimensions:
  - Number of data nodes (per CPU core)
  - Number of service nodes (per CPU core)
  - Size of chunk
  - Size of dataset
  - Use of compression
  - Cache hit vs. cache miss

- Is it possible to index source datasets using user supplied parameters. Instead of indexing a source file based solely on the chunked objects, can chunks be intelligently combined by HSDS when the underlying file structure supports it?

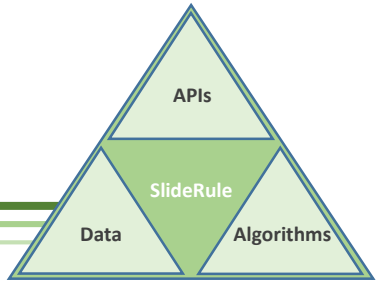


BACKUP



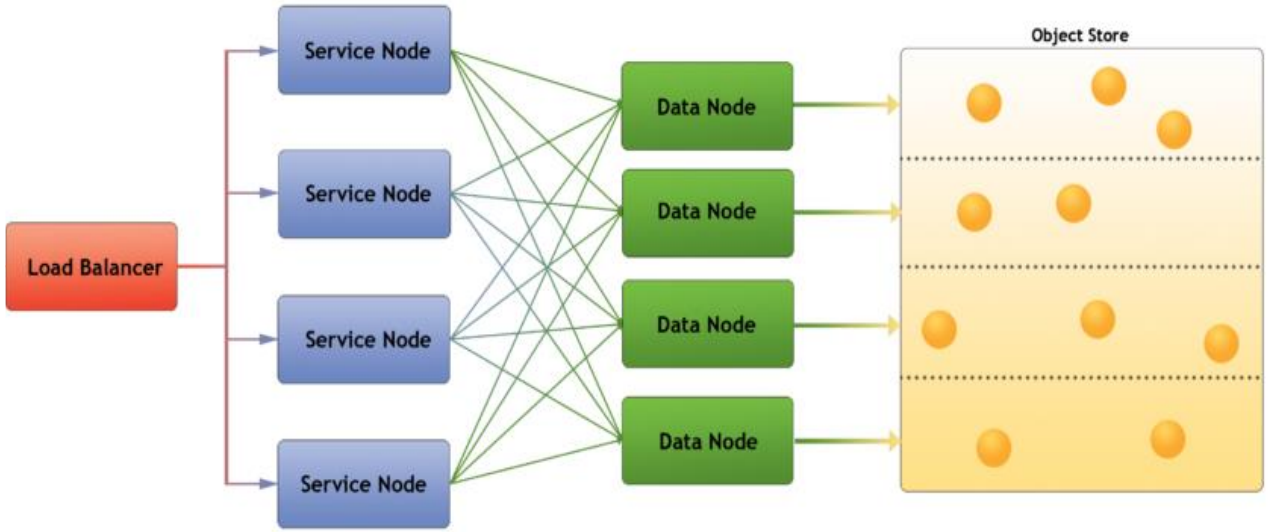
What is HSDS

HSDS Data Format



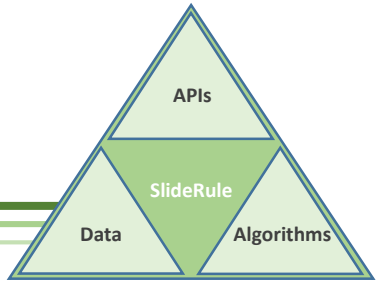
What Is HSDS

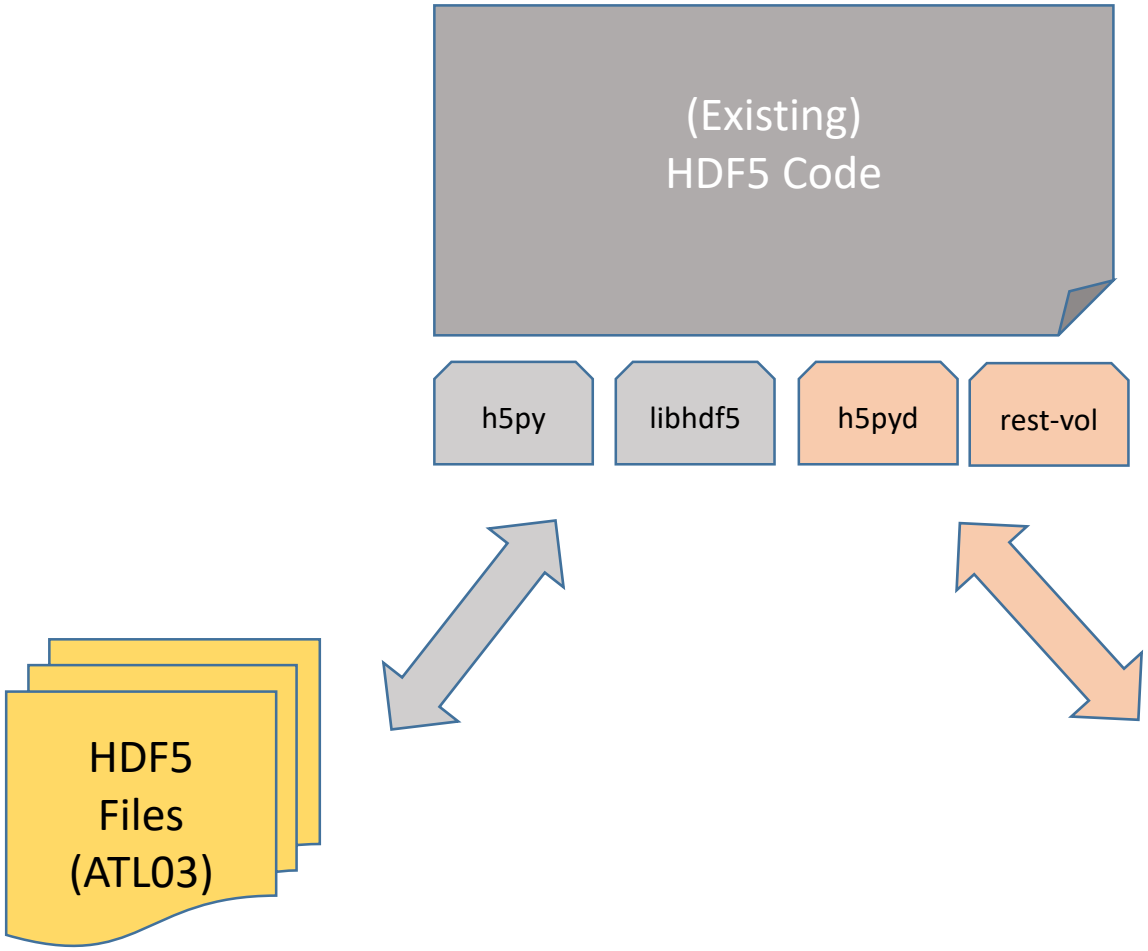
HSDS is a web service that implements a REST-based web interface for HDF5 data. Data can be stored in either a POSIX files system, or using object based storage such as AWS S3, or Azure Blob Storage. HSDS can be run on a single machine using Docker or on a cluster using Kubernetes (or AKS on Microsoft Azure)

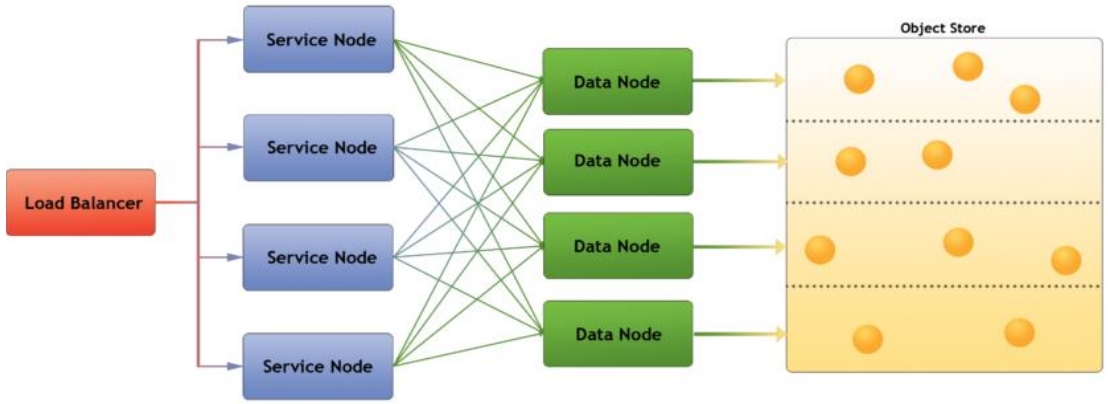


SlideRule uses HSDS to re-host the existing ICESat-2 datasets in a cloud-optimized format without reformatting the data and without breaking any existing tooling.

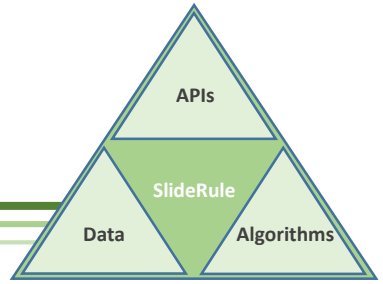
Highly Scalable Data Service (HSDS)

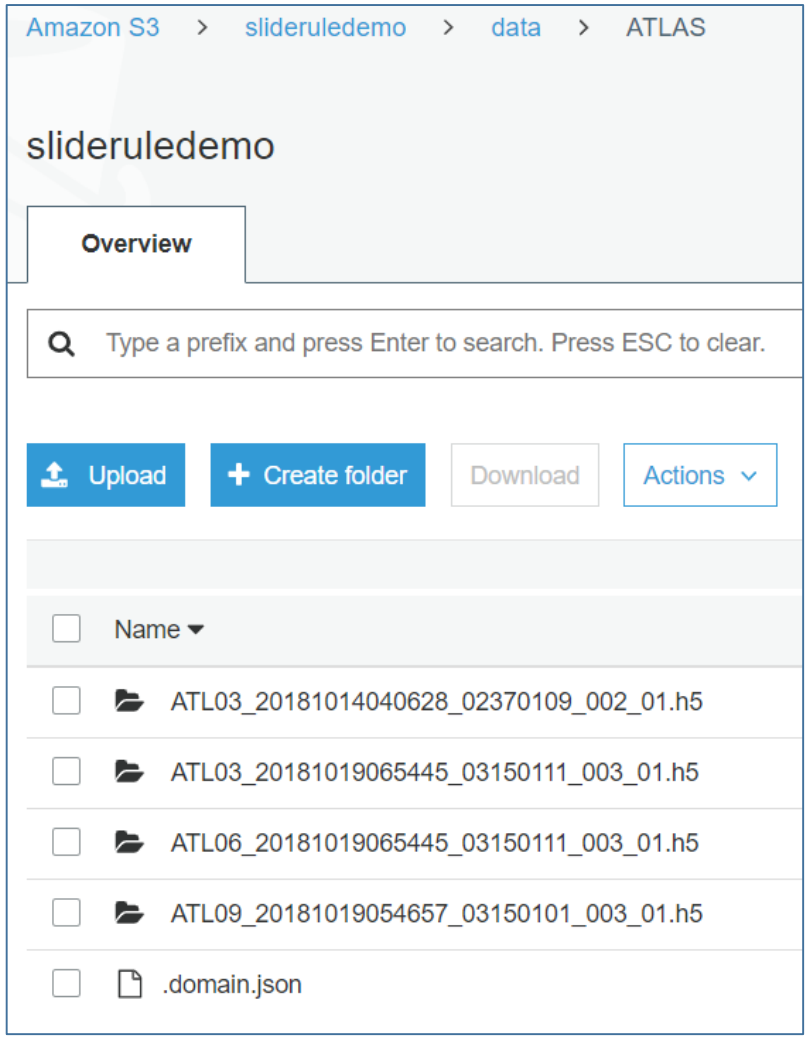




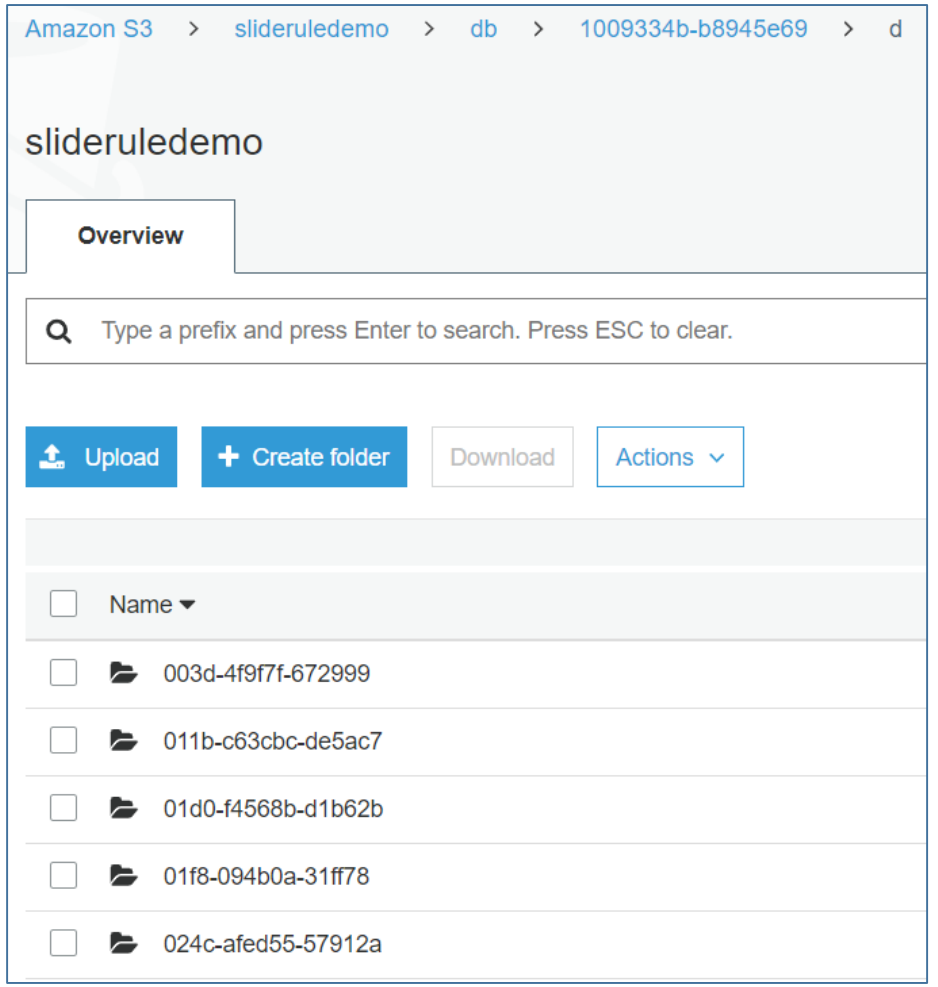


HSDS in S3

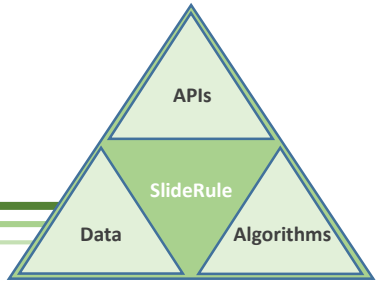




Each h5 file is an object that points to a sets of chunked objects in the same bucket.

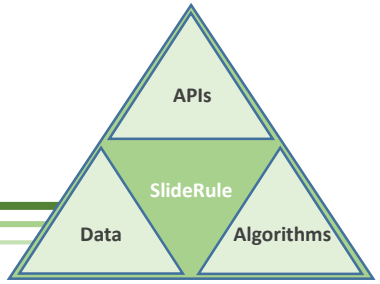


REST-VOL Additional Code



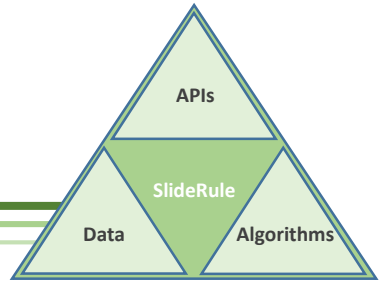

```
185  /*-------------------------------------------------------------------------
186   * init
187   *-------------------------------------------------------------------------*/
188  void H5Lib::init (void)
189  {
190      #ifdef H5_USE_REST_VOL
191          setenv("HSDS_ENDPOINT", DEFAULT_HSDS_ENDPOINT, 0);
192          setenv("HSDS_USERNAME", DEFAULT_HSDS_USERNAME, 0);
193          setenv("HSDS_PASSWORD", DEFAULT_HSDS_PASSWORD, 0);
194
195          H5rest_init();
196          rest_vol_fapl = H5Pcreate(H5P_FILE_ACCESS);
197          H5Pset_fapl_rest_vol(rest_vol_fapl);
198      #endif
199  }
200
201  /*-------------------------------------------------------------------------
202   * deinit
203   *-------------------------------------------------------------------------*/
204  void H5Lib::deinit (void)
205  {
206      #ifdef H5_USE_REST_VOL
207          H5Pclose(rest_vol_fapl);
208          H5rest_term();
209      #endif
210  }
```

Creating this file access property list and associating it with the rest-vol connector is the only change to existing C code needed to work with HSDS



Recommendations

Bindings

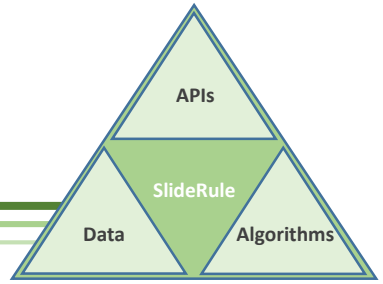


Currently, Python is the only fully supported language for interfacing with HSDS. The Python client is provided through the h5pyd Python package developed and maintained by the HDF group.

The rest-vol connector is the only other client available for talking to HSDS. It provides an excellent interface to HSDS and works well with existing C/C++ applications, but is no longer actively being developed.

Ideally, future efforts to improve HSDS will include supported clients in other languages like Java, Rust, etc. In addition, having the rest-vol connector come back as an actively maintained project would reduce the risk to existing efforts that have integrated with it.

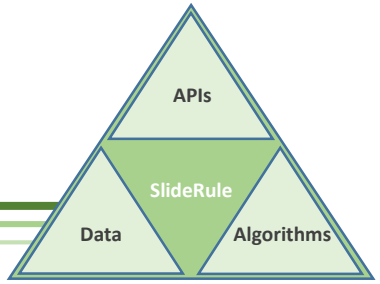
Serverless



Currently, in order to access h5 files in S3 using the HSDS interface, an HSDS server must be running.

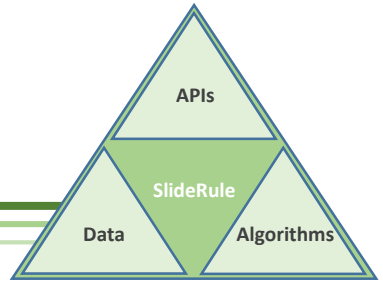
Servers cost money and must be maintained.

Ideally, the Python client (h5pyd) or some other client would be able to access the HSDS metadata directly (wherever they are stored) and perform the efficient reads of the h5 files in S3 without the need for an HSDS server running.



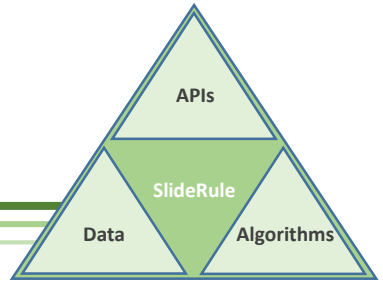
A Case for Cloud

Example Research Goal



Say you want to use ICESat-2 data to estimate the amount of snow that fell in 2019 in order to verify an existing model you've built using other data sources.

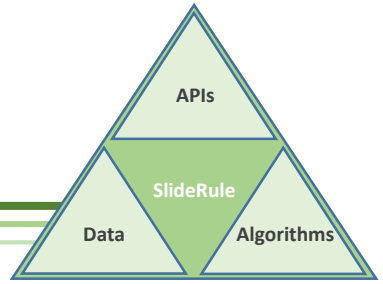
Problem #1



There is no higher-level data products provided by NASA that are optimized for snow-depth recovery.

As a result, you've got to go to the ICESat-2 point-cloud and atmospheric data (i.e. the ATL03 and ATL09 datasets).

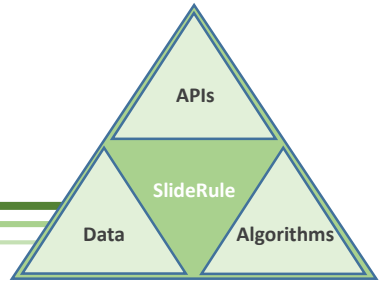
Problem #2



One year of ATL03 and ATL09 data is about 150TB.

Assuming an egress of 80Mbps, if you start pulling the data today, after 6 months of continuous, uninterrupted data transfer, you should have the 2019 data on your local computer.

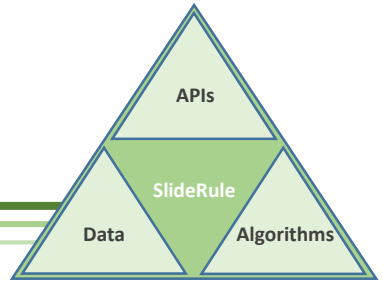
Problem #3



150TB doesn't fit on your local computer.

So now you go to your IT department (if you have one… if you are a small organization, you'll be doing this yourself), and ask for 150TB of storage you can NFS mount. The IT group goes out and buys 5 new machines, each with 4x10TB WD spinners in a RAID 5 configuration, and puts them on a 10GbE network.

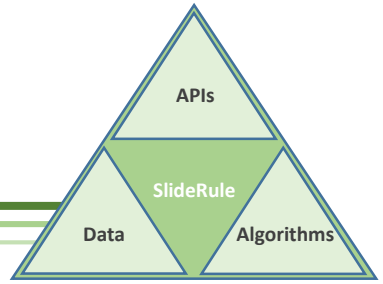
Problem #4



Even running on the fancy IT department servers, your Python code can only process 10MB/s.

So now you google MPI or Dask, choose one, get up to speed, rewrite more code than you expected, clean up a few choke points, and kick things off. It still takes about 2 weeks to complete one run; so you get some Python consulting help, do some further targeted optimizations, and you get it down to 5 days.

Problem #5

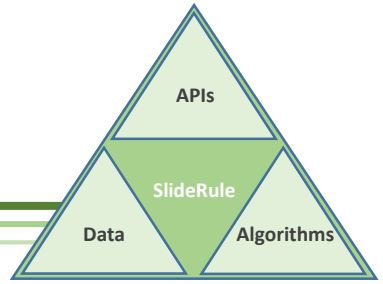


You succeed.

It’s six months later and everything is now downloaded; your code is ultra-optimized; and you've got a cluster at your disposal. Then your colleague at an out-of-state university has a few ideas and wants to collaborate.

How do you even begin to efficiently replicate and then build off of what you've done. Three months later you become the service part of “data-as-a-service” because everyone is asking you to run their ideas on your system.

Problem #6



A new version of the data comes out.