Web Accessible API in the Cloud Trade Study (Task 28)

EED2-TP-028, Revision 01

Technical Paper
August 2017
Prepared Under Contract NNG15HZ39C

RESPONSIBLE OFFICE

8/17/17

Alexander Schuster, Task Lead – EED-2 Task 28 — Date
EOSDIS Evolution and Development 2 (EED-2) Contract

RESPONSIBLE AUTHORS

8/17/17

James Gallagher, President, OPeNDAP, Inc. — Date
EOSDIS Evolution and Development 2 (EED-2) Contract

8/17/17

Ted Habermann, Principal Investigator — Date
EOSDIS Evolution and Development 2 (EED-2) Contract

Raytheon Company
Riverdale, Maryland

Task 28:
Web Accessible APIs in the Cloud Trade Study

James Gallagher[1], Ted Habermann[2], Aleksandar Jelenak[2], Joe Lee[2], Nathan Potter[1], and Kent Yang[2]

1. OPeNDAP Inc., 2. The HDF Group

August 2017

Table of Contents

Study Goals and Summary

Introduction

The Earth Science Data and Information System (ESDIS) Project manages the science systems of the Earth Observing System Data and Information System (EOSDIS). EOSDIS provides science data to a wide community of users for the National Aeronautics and Space Administration's (NASA) Science Mission Directorate. As part of its objective of continuously improving the level of service and evolving capabilities, the ESDIS project is looking to enhance the discoverability, accessibility, and usability of its data sets and services by focusing on open standards, web accessible application programming interfaces (APIs), and improved metadata. The role of cloud computing in facilitating these improvements is an important focus of several studies and on-going exercises.

One of the most popular web-based data services for ESDIS users is the Open-source Project for a Network Data Access Protocol (OPeNDAP), a data service that includes servers, clients, and the Data Access Protocol (DAP). OPeNDAP services are heavily used in traditional data center environments that include servers and datasets stored in file system storage. Most of the source data for these services is written in the Hierarchical Data Format (HDF) HDF5.

Providing equivalent data services in a cloud environment is critical for successful migration to the cloud. This study compares several potential architectures for OPeNDAP data services in the cloud using existing ESDIS datasets and representative Use-Cases. This report describes and presents results from that work.

Objectives

This study explored three candidate architectures for serving NASA Earth Science HDF5 data via Hyrax running on Amazon Web Services (AWS). We studied the cost and performance for each architecture using several representative Use-Cases.

The objectives of this project are:

- Conduct a trade study to identify one or more high performance integrated solutions for storing and retrieving NASA HDF5 and Network Common Data Format Version 4 (netCDF4) data in a cloud (web object store) environment. The target environment is Amazon Web Services' (AWS) Simple Storage Service (S3).
- Conduct needed level of software development to properly evaluate solutions in the trade study and to obtain required benchmarking/metrics for input into government decision of potential follow-on prototyping.
- Develop a cloud cost model for the preferred data storage solution (or solutions) that accounts for different granulation and aggregation schemes as well as cost and performance trades.

Terms

Granule: For this report, the term *granule* refers to an HDF5 file. Each granule is referenced by a single DAP Universal Resource Locator (URL).
Variable: A named array of values of the same data type. In DAP these are called variables, too. In HDF5 these are called *datasets*. Every granule (a.k.a. DAP URL) has zero or more variables. Every variable has zero or more attributes.
Chunk: When the HDF5 library is used to write array data, the array is broken into one or more chunks. Each chunk holds data from a different part of the array (see section HDF5 Library for further explanation).
A1: Architecture 1
A2: Architecture 2
A3: Architecture 3

There is also a Glossary on page 41.

Assumptions

Assumptions we made during this study:

a. The Data Access Protocol (DAP) was not changed to accommodate any of the architectures, so no new client behavior is needed when accessing data from S3.
b. We are using Hyrax on Amazon Elastic Compute Cloud (EC2) running CentOS 6 instances for data access.
c. To focus on actual S3 behavior and performance, we will minimize the effects of any caching. Thus, evaluating the effect of caching, cache sizes, caching strategies, or cache ejection polices is not part of this study.
d. The EC2 instances for the Hyrax server and the data in S3 are collocated in the same AWS region.
e. We are using publicly available AWS price list for calculating costs.
f. Cost computations are based on our best understanding of the actual AWS billing system.

Findings and Recommendations

Three architectures were compared in this study for a variety of use cases. Highlights are compared in Table 1.

- Architecture 1 (A1) was roughly five to ten times slower than Architectures 2 (A2) and 3 (A3) for requests in which a single variable was requested from each granule.[1]
- However, when several variables (30 variables for the Atmospheric InfaRed Sounder (AIRS) data -- 17 for Modern-Era Retrospective analysis for Research and Applications Version 2 (MERRA2) -- were requested from each granule, or all the variables in the granule were requested, A2 was from 1.6 to 2.6 times slower than A1, and A3 was between 2.6 and 3.7 times slower than A1.

[1] In the DAP, requests are always for one or more variables. For both the AIRS and MERRA2 granules used in this study, each variable uses more than one chunk, so requesting a single variable requires A2-3 make several requests to S3.

- Architectures 1 and 2 both have identical data storage, so an adaptive approach that combines these two would make excellent sense. Using some metric, the server would decide to retrieve individual chunks or collect the entire granule from S3.
- Data storage costs for A3 can be less for some granules. In those cases, A3 may be preferred.

Our recommendation is to develop a hybrid of A1 and A2 unless specific granule characteristics indicate significant storage savings of A3.

Table 1: Overview of the three architectures examined in this study.

	Architecture 1	**Architecture 2**	**Architecture 3**
Performance	Faster than A2 & A3 for requests that access many variables or the entire granule. Requests for one or two variables are much slower than for A2 & A3.	Faster than A1 for requests accessing one or two variables. Slower for requests for many variables or the entire granule.	Faster than A1 for requests accessing one or two variables. Slower for requests for many variables or the entire granule.
Processing Costs	Depends on processing time	Depends on processing time	Depends on processing time
Storage Costs	~equal to A2	~equal to A1	Can be 1% to 30% lower depending on repetition of data values within the granule.
Full granule retrieval	Yes	Yes	No
Data Migration to Cloud	Copy each granule file to a single S3 object	Copy each granule file to a single S3 object	Shred each granule file into multiple S3 objects
Commercial Web Crawler Access (GoogleBot, etc.)	Potentially significant if crawlers require large amounts of information to move from S3 to the data server. This situation can be mitigated by limiting crawler access to just metadata held by the server.		

The results of this study indicate that the principal cost drivers are:

1. The time required to transfer data from S3 to the Hyrax server running on EC2.
2. Storage of data on S3 (data and metadata).
3. Egress of data from AWS to users. Even though we factored this cost out of our analysis, it is obvious that this cost is a significant component of the overall cost.

Our findings that are relevant to these costs:

1. It is possible to decrease processing costs by minimizing waiting for data. This would mean implementing code that would evaluate the request and based on a metric utilize either the A1 or A2 approach to minimize the data retrieval time.

2. It is possible to decrease data storage costs by identifying and avoiding storage of redundant chunks (~30% savings in A3 for our sample AIRS data), but this comes at the loss of access to the original source HDF5 granule.
3. Egress costs (bytes leaving AWS) are the most significant costs (~4x storage costs). These costs depend on user behaviors and are equivalent for the three architectures we explored.

Other findings include:

1. The variables in the tested HDF5 granules had a significant number of “duplicate” chunks (chunks with identical content), and we leveraged this when we shredded the HDF5 granules for A3. We stored duplicate chunks only once, thus saving storage space.
2. While the HDF4 chunks representation that we leveraged includes chunk offsets, sizes, unique identifiers and checksums, the Universally Unique Identifier (UUID) turned out to be unnecessary as it was superseded using the Message Digest Version 5 (MD5) value (and will be replaced by Secure Hash Algorithm 2 (SHA2) checksums in the future). The remaining parameters were adequate for the retrieval of chunks using Hypertext Transfer Protocol (HTTP) range requests against a single object or for retrieval of entire S3 objects.
3. Given the expense of data accesses and typical DAP client behavior, expanding on the granule-level metadata stored outside of S3 will likely reduce costs. The additional metadata should focus on ‘use’ metadata such as the ranges of values in the granule’s variables.
4. Our National Snow and Ice Data Center (NSIDC) log analysis indicates that ~96% of their opendap requests are associated with general purpose web crawlers. (We identified 22 distinct automated crawlers.) We suspect that crawler traffic is similarly high at the other Distributed Active Archive Centers (DAACs), but we could not confirm that suspicion without user agent information. We need to identify a mechanism with which we can satisfy crawler requests (desirable in terms of web search and discovery) while discouraging them from costly activities. It may be that NASA’s recent migration to authenticated access for data has addressed this concern. The log files we reviewed included dates prior to NASA’s change in access control.
5. Inconsistencies across server logs from the various DAACs complicated log ingestion and limited the scope of what we learned about how clients utilize the servers. Five fields are required in logs to fully characterize the ways in which clients utilize the DAP service:
    a. Date & Time (with time resolution of 1 second or better)
    b. Hashed Client internet protocol address (preferably with SHA2)
    c. Request (URL path + query string)
    d. Response Size
    e. Requester’s User Agent

Recommended Activities

Integrate support for S3 into the HDF5 Library. The current implementation of A2 and A3 retrieves data from granules using direct access to data from S3. To develop this proof-of-concept code quickly, we re-implemented some capabilities of the HDF5 library in the Hyrax data server. It would be better to build support for S3 into the HDF5 library.

Refine the implementations of Arch 1, 2 and 3 in Hyrax. The implementations of Architectures 1, 2 and 3 in Hyrax are not 'production quality' software; refine these implementations, since the effort to move these to production will be minimal. The current implementations support parallel access to S3, but this needs to be optimized. Use the HDF5 library support for S3 when it becomes available.

Develop an adaptable server. Instead of choosing one of the three architectures, develop an implementation that chooses different behavior based on the characteristics of the data and the request. We know from experimentation that the different architectures respond more effectively to different kinds of requests. It is possible to merge A1 with the A2/A3 implementations to produce a single adaptable server.

Model and mitigate web crawler costs. Since many accesses to the DAAC's DAP servers come from web crawlers, we should develop a more concrete description of crawler behavior and a way to mitigate the costs of crawler traffic. At the same time, the mitigation strategy should not limit the information those crawlers extract because they may provide important data discovery services for users.

Model current DAAC data use. To determine the likely actual cost of moving large volumes of data from a DAAC's on premises storage to the AWS cloud, develop a model, based on measured use, of that DAAC's DAP server. This will enable us to forecast the cost and resources needed to satisfy the current level of demand for data. This same model can also compare the performance of cloud systems to the current performance of DAP services.

The logs that we received from the DAACs were inconsistent and did not include information required for clear and unambiguous understanding of how OPeNDAP is being used (see Finding 6 above and discussion below). We need to improve consistency across the DAACs and ensure that important information is included in the logs.

Explore deployment utilizing a serverless architecture. Evaluate the cost and performance of a 'serverless' implementation of the DAP web services relative to the current version that runs on an EC2 instance. An AWS 'serverless' implementation uses newer features of AWS that eliminate the need to run a traditional web server on a virtual computer when data are not being accessed, thus eliminating the cost of an idle EC2 instance. Adopting this may incur performance penalties, however.

Performance Comparison of the Three Architectures

Table 2 summarizes representative performance results comparing all three architectures for twenty-one Use-Cases run during several time periods. The total times for running all Use-Cases are shown in the first two rows. These times are most representative of a typical set of DAAC access Use-Cases, and they indicate that the three architectures each have advantages in certain request patterns. For requests that touch a few chunks (one or two variables), and thus make only a few HTTP GET requests to S3, A2 and A3 are roughly ten times faster than A1. However, when more variables (e.g., 17 MERRA2 variables or 30 AIRS variables) are requested from a granule, the A2 and A3 code takes longer than the A1 code to produce a response (1.7 to 3.2 times more than the A1 time). When all the variables in a granule are re-

trieved, A2 and A3 take 3 to 6 times longer to service the request than A1. The important discriminant in these two kinds of access is not actually the number of variables, but rather the number of chunks. Each chunk is accessed using a separate HTTP GET request. Minimizing the number of HTTP GET requests made by the server to S3, all other things being equal, improves performance. The results for a variety of specific Use-Cases are shown in the rest of the Table. They support the overall performance differences.

Table 2: Performance gain and loss using Architecture 1 as baseline. The smaller the percentage the better performance.

Date	Test	A1	A2	A3
February	**All Use-Cases**	**1244 minutes**	**111 minutes**	**108 minutes**
March	**All Use-Cases**	**1140 minutes**	**100 minutes**	**108 minutes**
2/26/17	Use-Case 2 Dataset Metadata Response (DMR)	100.00%	8.74%	8.14%
2/26/17	Use-Case 6 MERRA2 aggregation	100.00%	19.98%	16.45%
2/26/17	Use-Case 7 aggregation A	100.00%	4.05%	3.80%
2/26/17	Use-Case 10 aggregation A	100.00%	5.49%	4.99%
2/26/17	Use-Case 11 aggregation A	100.00%	10.40%	12.43%
2/26/17	Use-Case 12 aggregation M	100.00%	26.43%	23.79%
2/26/17	Use-Case 13 aggregation M	100.00%	48.23%	57.03%
2/26/17	Use-Case 14 aggregation M	100.00%	16.49%	15.47%
2/26/17	Use-Case 15 aggregation A	100.00%	4.86%	4.53%
2/26/17	Use-Case 16 aggregation M	100.00%	86.21%	83.20%
2/26/17	Use-Case 17 aggregation A	100.00%	10.03%	10.00%
3/5/17	Use-Case 2 DMR	100.00%	9.07%	8.42%
3/5/17	Use-Case 6 aggregation M	100.00%	20.91%	16.45%
3/5/17	Use-Case 7 aggregation A	100.00%	4.25%	3.79%
3/5/17	Use-Case 10 aggregation A	100.00%	5.70%	5.11%
3/5/17	Use-Case 11 aggregation A	100.00%	11.30%	12.82%
3/5/17	Use-Case 12 aggregation M	100.00%	27.21%	24.29%
3/5/17	Use-Case 13 aggregation M	100.00%	51.09%	59.84%
3/5/17	Use-Case 14 aggregation M	100.00%	16.87%	15.91%
3/5/17	Use-Case 15 aggregation A	100.00%	4.72%	6.20%
3/5/17	Use-Case 16 aggregation M	100.00%	32.95%	30.24%
3/5/17	Use-Case 17 aggregation A	100.00%	10.58%	10.80%
3/14/17	Use-Case 18 multivariable multi-file M	100.00%	202.71%	316.11%
3/14/17	Use-Case 19 multivariable multi-file A	100.00%	169.47 %	278.02 %
3/14/17	Use-Case 20 all variables multi-file M	100.00%	433.89 %	597.36 %
3/14/17	Use-Case 21 all variables multi-file A	100.00%	287.33%	419.11%

Cost Comparison of the Three Architectures

There are two general contexts in which the architectures can be compared: performance-driven and performance-agnostic. In both contexts, we will assume the same client (DAP) requests are made which will also result in the same amount of egress data and the related costs. In the performance-driven context, the time to complete client requests (same work) directly impacts the total costs. This is relevant for cases where providing DAP web service is required either to increase the capacity or support specific data processing effort for a limited period. The performance-agnostic context is applicable to operational availability (constant presence) of DAP web service.

Performance-Driven Context

The comparison presented here is based on the cost data reported when running the use cases developed for this study. The cost data were collected from the Detailed Hourly AWS Cost and Usage Reports, one of the available types of AWS billing reports.

The only cost data altered from the original was related to the S3 storage of the sample HDF5 data. There were two reasons:

1. We stored both the original HDF5 granule files (A1 and A2), and chunks (A3) in the same S3 bucket, and thus the reported S3 storage costs did not reflect the storage savings (due to reduced storage size) for A3 versus A1 and A2.
2. We noticed that S3 storage costs are calculated daily and saved as a single hourly cost entry once per day in the reports. This caused the S3 storage costs to be higher than would be realistic for the Use-Cases' runtimes.

We computed new S3 storage costs using the storage data in Table 4 and for the duration of the Use-Cases execution for each of the three architectures.

Processing Costs

Table 3 shows the overall processing cost comparisons for running the three Architectures. The only significant difference between the three is related to the hourly usage charge for the EC2 instance and the attached Elastic Block Store (EBS) storage during the long duration of the Architecture 1 runs (~19 hours vs. two hours).

Table 3. Processing cost comparisons.

Architecture	1	2	3
AmazonEC2	**$13.572**	**$1.721**	**$1.723**
BoxUsage:m4.xlarge	$12.685	$1.720	$1.720
DataTransfer-Regional-Bytes	$0.001	$0.001	$0.003
EBS:SnapshotUsage	$0.065		
EBS:VolumeUsage.gp2	$0.821		
AmazonS3	**$0.103**	**$0.025**	**$0.022**
Requests-Tier2	$0.008	$0.015	$0.014
TimedStorage-ByteHrs	$0.095	$0.010	$0.008

S3 Storage Costs

The three architectures consume different amounts of S3 storage. Table 4 shows the storage usage in gigabytes (GB) for serving the data used in this study. Since A2 and A3 require the index (DMR++ eXtensible Markup Language (XML) files, an additional 2.1 gigabytes were added to the S3 storage amount totals for these two architectures so that the Hyrax server would be able to serve incoming requests.

Table 4. The amount of stored data in S3 for the different architectures.

Architecture	1	2	3
Storage Requirement (GB)	156	158.1	126.1

The storage required for Architecture 1 and 2 are very similar. The decrease in storage by ~20% for A3 reflects the savings by having only one copy of duplicated chunks.

S3 Data Requests

The total number of actual S3 requests for the Use-Cases is shown in Table 5. In Architectures 2 and 3 the number of S3 requests from the Hyrax server is larger when compared to Architecture 1 because there is a request for each chunk rather than each granule. The difference between A2 and A3 is due to some use cases that involved random array sub-setting requests and, therefore, different numbers of chunks.

Table 5: Total number of S3 requests from all the Use-Cases for each architecture.

Architecture	Number of Requests	Cost
A1	19,709	$0.0079
A2	37,152	$0.0149
A3	34,592	$0.0138

Despite the significant difference in numbers of S3 requests, the cost difference is small because S3 requests are one of the least expensive AWS resources (currently at $0.04 for 100,000 requests) so they do not contribute significantly to the total cost.

Performance-Agnostic Context

Cost comparison of the three architectures in this context entails that the only cost differentiators are the number of S3 requests from the Hyrax server (EC2 instance) and the amount of data stored in S3. The EC2 instance is assumed to run constantly (24/7) and that DAP requests are the same and, hence, produce the same DAP responses (same amount of data egress).

S3 Storage Costs

The cost situation is unchanged from the previous context. A3 can reduce this cost but it depends on the data itself so the savings cannot be predicted without inspecting the data.

S3 Data Requests

The number of S3 requests for one DAP request varies based on the architecture and the DAP request's type. In A1, one DAP request can cause zero or one S3 requests. For the other two architectures, DAP data requests may generate 0 or more S3 requests depending on the number of variables, sub-setting expressions in the request, and the number of chunks that must be read from. Since DAP requests cannot be predicted, the question becomes: can there be enough DAP requests to generate such a large number of S3 requests that they constitute a significant portion of the total cost?

The S3 requests' cost is the least expensive of all AWS services we observed during our study, at $0.04 for 100,000 requests. As an example, just one hour ($0.215) for the EC2 instance used in this study equals 537,500 S3 requests. Or, the price of 1 gigabyte of data per month egressing the AWS cloud ($0.09 for the first 10 terabytes) is equal to 225,000 S3 requests per month. Downloading just three granules from one NASA product from this study is almost equal to one gigabyte.

To answer our question, the following is an example with the AIRS product (NASA, 2017) used in this study. This AIRS product has 777 variables per granule and a total size of approximately 1,643 GB in 5,300 granules. The maximum number of chunks per AIRS variable is eight. If the DAP requests ask for all the variables in all the granules, the maximum number of S3 requests possible is: 777 × 8 × 5300 = 32,944,800. The cost of all these requests is: 32,944,800 × $0.0000004 = $13.18. This amount converted to egress data equals $13.18 / ($0.09/GB-mo) = 146.42 GB per month. Since the DAP requests are asking for nearly the entire 1,643 GB data collection, the size of data egress would be very close to this amount. This means the cost of the S3 requests is equivalent to 8.9% of the egressed data they produced. In other words, the cost of S3 requests will always be eclipsed by other costs.

Study Methodology

Sample Data Collections

We considered three different test data collections:

1. NASA AIRS Standard Retrieval product (365 granules; 109G)
2. NASA MERRA2 products (7,466 granules; 231G)
3. Sample HDF5 granules (96 granules; 125MB)

As discussed below, the behavior of these test collections in the cloud depends critically on several characteristics that are not necessarily apparent on cursory examination. The important characteristics determine how the data collections will behave in a realistic ESDIS data service.

For example, if the granules include a small number of variables, users may tend towards downloading the entire granule for simplicity or because they are interested in all of the variables. Conversely, if the granule has many variables, users may be particularly interested in subsets in

the variable space for certain science questions; for example, they may only be interested in sea surface temperature data. The behavior is also related to the patterns used for chunking the variables. If the variables are contiguous (I.e. one chunk), then the entire variable can be accessed with a single call to S3. If the variables are chunked, the number of S3 retrievals required to access the entire variable will be the number of chunks in the variable.

The NASA AIRS test data we utilized includes 365 Version 6 daily granules from 2015 that were converted from HDF4 to HDF5 for this study. Each of these granule contain 777 variables which collectively represent 569,765 chunks (~1,561 chunks / granule) of which 330,353 (58%) are unique. The most common redundant chunk occurs 49,640 times. The redundant chunks make up 29.4% of the total variable data. Not storing the redundant chunks thus saves significant fixed storage costs.

The MERRA2 collection includes many different products. We focused on 440 granules of the MERRA2_100.tavgM_2d_slv_Nx product to test the data aggregation Use-Cases. These granules include 197 variables that are stored in one or sixteen chunks. Collectively these granules included 82,639 chunks (~chunks/ granule) of which 81,057 are unique. The most common redundant chunk occurs only 108 times. In this case, redundant chunks make up less than 1% of the data volume.

The sample HDF5 granules are a miscellaneous collection that have been accumulated over years of interactions. We included these granules in case we needed to test particular data structures.

HDF5 File Content Map - Index Files

Many existing web data access architectures include catalogs or index files that separate the logical organization of the data from the physical data storage and include additional metadata that are required to support access services (e.g., Thematic Real-time Environmental Distributed Data Services (THREDDS) (Unidata, 2017), Globus (Globus, 2017), OPeNDAP (NOAA, 2017)). Typically, the smallest data item included in these catalogs is a granule (i.e., a file). This approach works well if the architectures retrieve entire granules to service user requests (e.g., file transfer protocol or A1, described below), but they are not helpful in architectures where sub-sets of granules (e.g., only 3 of 500 variables from a granule) are requested from the resource stored in S3 or similar object stores.

To our knowledge, the only existing catalog that includes sub-granule information for HDF data is the HDF4 Map representation developed by ESDIS and The HDF Group (The HDF Group, 2017) to provide sustainable access to HDF4 data if the HDF4 libraries were no longer available or supported. The maps are written in eXtensible Markup Language (XML) and include critical details, e.g., sizes and offsets, required to access and decompress data objects in HDF4 granules shown below.

The DAP4 protocol uses the Dataset Metadata Response (DMR) to transmit the semantic, structural description, and metadata data content of a data source (OPeNDAP, Inc., 2007). The DMR representation includes metadata information characterizing the variables in a granule, their datatypes, names and attributes.

To provide indexing support for Architectures 2 and 3, we combined HDF4 Map's XML representation for sub-granule data items with OPeNDAP's DMR representation of metadata content and structure to form a representation that we termed DMR++. A DMR++ representation for the `GPHeight_MW_A_sdev` variable in one of the AIRS granules is shown in Table 6. It includes existing DMR information about dimensions and attributes and new information about chunks (tagged as `h4:chunks` and `h4:byteStream` elements).

Table 6: A DMR++ representation for the GPHeight_MW_A_sdev variable in one of the AIRS granules.

```
<Float32 name="GPHeight_MW_A_sdev">
    <Dim name="/StdPressureLev"/>
    <Dim name="/Latitude"/>
    <Dim name="/Longitude"/>
    <Attribute name="_FillValue" type="Float32">
      <Value>-9999.</Value>
    </Attribute>
    <h4:chunks deflate_level="2" compressionType="deflate shuffle">
      <h4:chunkDimensionSizes>12 90 180</h4:chunkDimensionSizes>
      <h4:byteStream uuid="6bf9e169-b7c1-4f89-a3a3-7bcfa5e5a78b"
                     nBytes="3421"
                     offset="67779707"
                     md5="c55324594a0787856d9b908fc7a7a201"
                     chunkPositionInArray="[0,0,0]"/>
      <h4:byteStream uuid="a2dfe411-3699-4007-8245-1da034f78af5"
                     nBytes="3421"
                     offset="67783128"
                     md5="c55324594a0787856d9b908fc7a7a201"
                     chunkPositionInArray="[0,0,180]"/>
      <h4:byteStream uuid="80dd97ed-f90e-43aa-a34c-bc4a8a545900"
                     nBytes="3421"
                     offset="67786549"
                     md5="c55324594a0787856d9b908fc7a7a201"
                     chunkPositionInArray="[0,90,0]"/>
      <h4:byteStream uuid="b7d992ab-aca4-463e-8465-909893d478a0"
                     nBytes="3421"
                     offset="317253596"
                     md5="c55324594a0787856d9b908fc7a7a201"
                     chunkPositionInArray="[0,90,180]"/>
      <h4:byteStream uuid="b14adf06-d769-4298-800a-a4e79ab948dd"
                     nBytes="3421"
                     offset="317257017"
                     md5="c55324594a0787856d9b908fc7a7a201"
                     chunkPositionInArray="[12,0,0]"/>
      <h4:byteStream uuid="dc4ecfc9-9899-4266-90dd-909d332cde0a"
                     nBytes="3421"
```

```
                    offset="317260438"
                    md5="c55324594a0787856d9b908fc7a7a201"
                    chunkPositionInArray="[12,0,180]"/>
      <h4:byteStream uuid="3419b454-a29b-4cc6-8d8c-e2553f321d41"
                    nBytes="3421"
                    offset="317263859"
                    md5="c55324594a0787856d9b908fc7a7a201"
                    chunkPositionInArray="[12,90,0]"/>
      <h4:byteStream uuid="1391743f-57c2-4a4e-865a-599ab4ef6f77"
                    nBytes="3421"
                    offset="317267280"
                    md5="c55324594a0787856d9b908fc7a7a201"
                    chunkPositionInArray="[12,90,180]"/>
    </h4:chunks>
</Float32>
```

The `h4:chunks` element includes information about compression and chunk dimensions as well as a list of `h4:byteStream` elements encoding the position of the data in the granule (`offset`) and the size of the data (`nBytes`) of each chunk. Several attributes were added to the HDF4 Map chunk representation to support access to data in the cloud: the chunks were given universally unique identifiers (UUID) and had MD5 checksums calculated based on their byte content to address data integrity concerns and requirements expected to be associated with the project's plan to engage in a data shredding processes to produce data to be utilized in A3.

Creating the DMR++ files for our sample data required development of three new capabilities:

1. Extraction of the chunk (`h4:byteStream`) information from HDF5 files, and
2. A tool for reading existing DMR and HDF5 files and augmenting them with the byte stream storage information.
3. Added a URL to the relevant S3 object as an attribute to the DMR++'s Dataset element in A2, and added a URL to the S3 object as an attribute to each h4:byteStream element in A3. This allowed the DMR++ files to be completely portable from one system to another without "loosing" their reference to the binary data.

The first capability was added to a new custom branch of the HDF5 library and implemented in the h5py library for access using Python. The second capability was developed as a prototype stand-alone tool written in Python that utilized the first capability. And the third was accomplished using a bash script.

Use Cases

To characterize Hyrax cost and performance in the cloud, we identified several representative use cases that could be tested across all three architectures based on common usage found in Hyrax server logs from NASA DAACs. Initially, these use cases were designed to include two configuration options of the OPeNDAP HDF handlers (Climate Forecast [Convention] (CF) and default) and two versions of the DAP Protocol (DAP2 and DAP4). All options and protocols

could be tested in A1. As the other architectures were implemented, we focused on the use cases using the CF configuration option of the HDF5 handler and DAP4 protocol because we anticipate that will be the primary access path used by NASA data users.

Analysis of Hyrax Logs from NASA Data Centers

We collected Hyrax request logs from the Goddard Earth Sciences Data and Information Services Center (GES DISC), Atmospheric Science Data Center at NASA Langley Research Center (LaRC ASDC), National Snow and Ice Data Center (NSIDC), and Land Processes Distributed Active Archive Center (LP DAAC). In total, we collected logs of 18,409,186 user requests. To protect user privacy the source institutions replaced the original internet protocol addresses with anonymous hashed values. The ASDC log we received had the largest number of requests, with more than 10 million records and LP DAAC had the smallest, with 500,000. Some of our analysis of the logs was conducted using ElasticSearch and Kibana, other analyses were conducted using custom code to decompose and group the logs in various ways.

The fields in these logs varied across the DAACs. In the union of the logs we found a total of 22 unique fields, yet the logs had only 4 fields in common. This precluded several informative analyses we would have like to have made. For example, in the logs we received, only the NSIDC log provided user-agent information which is crucial to understanding the behavior of both crawlers/bots and of programmatic access clients. The time granularity varied quite a bit between the logs also:

- NSIDC – 1 second
- LaRC ASDC – 1 second
- LP DAAC – 1 minute
- GSFC GES DISC – 1 day

which also makes access patterns difficult to discern.

Some interesting patterns emerged from the examination of these logs:

1. Web crawlers (‘bots) are clearly responsible for most traffic in the NSIDC log (the only logs we had with user-agent field that allows the unambiguous identification of crawlers and robots). The ‘bot access is not limited to HTML and XML files. The ‘bots also make requests for DAP data and metadata responses. These ‘bots made 3,196,153 of the 3,327,182 requests in the NSIDC log, or roughly 96% of total accesses.
2. PyDAP and OC4.3.2 (netCDF library) are responsible for most the programmatic access at NSIDC (NSIDC logs were the only ones with User-Agent information)
3. In some cases, programmatic access looks for a specific spatial subset of data from multiple granules of the same product for different dates. This pattern was well represented in our Use-Cases.
4. In some cases, we identified very inefficient sub-setting/request behaviors. For example, one log showed that over 900,000 requests, each for a single value from a multi-dimensional array were submitted to the server. The same data could have been retrieved using a single request utilizing an OPeNDAP constraint expression to retrieve all of the values. Examining these behaviors in the context of logs with the additional user-agent information would allow us to understand these patterns in detail.

Use Case Tags

To unambiguously track performance and cost for each use case we added a special URL query parameter to each Hyrax request: cloudydap. This parameter was included in every DAP request and passed on by the Hyrax server to every AWS S3 request it made. Such requests were reported in S3 access logs, making it possible to connect each S3 request to the appropriate client DAP request. The format of the cloudydap parameter values was:

```
cloudydap = {UseCase}_{Arch}_STARTED_{seconds-since-epoch}.h5
```

where:

- `{UseCase}` is the Use-Case identifier, e.g. `UC1` for the Use-Case 1.
- `{Arch}` is the architecture identifier, e.g. `A1CFT` for A1 CF=True Hyrax server.
- `seconds-since-epoch` would be replaced with the output of a `date +%s` UNIX command (ex: `1485208202`) which should be the same for every request in a specific run of a collection of the Use-Cases.

Testing the Use-Cases

We created test scripts for each of the Use-Cases and ran these scripts weekly against the three different architectures. While some scripts modeled individual accesses against single granules, other scripts simulated the aggregation behavior of the Hyrax NetCDF Markup Language (NcML) handler (which was not available with the proof-of-concept code) using the Hyrax command line tool, Back End Server Command Line (bescmdln). The retrieval results were compared with results for the same requests run locally on a Hyrax server at The HDF Group.

System Architectures

Overview

To understand performance and cost of serving data in AWS, we implemented and compared three alternative architectures that differ in storage approach (complete granules vs. chunks) and access approach (complete granules vs. parts of granules vs. complete chunks). The architectures are summarized in Table 7 below and described in detail in Appendix 1. The first architecture is a baseline for estimating cost and performance. For the second and third architectures, we compared cost and performance to the first architecture.

Table 7: Summary of storage and access approaches for the candidate architectures.

Architecture	AWS Storage	AWS Requests	Extract Transform & Load Process
A1: Entire granule	Object = Entire granule	One HTTP GET request for each requested granule. Last granule accessed is cached.	Copy entire granule.
A2: HTTP range request	Same as A1*	One HTTP range GET request per chunk.	Same as A1.
A3: Object store	Object = Contiguous variable or one of its chunks	Same as A2.	Shred granule into contiguous variables and chunks.

*If HDF5 File Content Maps are stored in S3, storage cost will be more than A1.

Architecture Notes

During this study, we have noted several characteristics/features that differentiate these architectures from one another. These are described briefly here.

Migrating Data to the Cloud

Data must be migrated from existing storage to the cloud to be served from the cloud and the process used for this migration may impact ESDIS decisions so we describe it here. For A1 and A2 the objects stored in S3 are complete granules so transferring these data to the cloud is primarily a simple copy operation. We did this migration using a script that copies directory structures and files and creates an XML index file that emulates the original directory structure to simplify access for direct users of the S3 repository.

A3 requires the original granules to be *shredded* into S3 objects where each object contains a single variable (contiguous storage) or a single chunk of a variable. This requires knowing where these chunks are in the original data granules, extracting them from the original granules, transferring them to S3 and naming them. We chose to do this process from a remote source (The HDF Group internal server) rather than from granules already in S3 to emulate the process of a transfer from an existing DAAC. We wrote a tool that extracts information about chunk locations and sizes from DMR++ files and transferred the identified chunks to S3 as objects.

Naming Objects in the Cloud to Decrease Storage Costs

Many ESDIS data systems include elaborate schemes for naming data granules to support easy user recognition of products and, in many cases, time periods. These names can be preserved in the index files described above for A1 and A2, but the granules no longer exist in A3. As described above, we created unique identifiers for each data element in anticipation of using those identifiers as object identifiers but instead utilized the MD5 hashes to name the S3 objects utilized in A3.

As we shredded granules into objects we discovered that many of the data objects had the same MD5 checksums which indicates that they have identical content. (While MD5 may not be perfect in this regard it was enough for this study) This means that we can save space by storing only one instance of each of those chunks. One might expect that the number of repeating chunks would be small, but it turns out to be substantial in some situations. Our test data included 365 granules of the AIRS3STD.006 (NASA, 2017) product from 2015. That variable included four chunks that occurred more than 1000 times with the most common chunk occurring 49,640 times (136 times per granule). Eleven chunks occurred 365 times (once per granule). All told, removing these repeating chunks decreased the storage from ~115 GB to ~81 GB (29%), decreasing the number of S3 objects by 42%. In the MERRA 2 case, the size of repeating chunks was much smaller, less than 1% of the total. These decreases will decrease storage costs in a way that depends on specific characteristics of each variable.

Original HDF5 Granule Downloads

A1 and A2 preserve the original granules intact and, therefore, those original granules are available for download. This is not true for A3. *Once a granule is shredded, it is not straightforward, and probably not possible, to reconstruct the original granule file*. However, data would be available in several formats including as a downloadable netCDF-4/HDF5 file by utilizing Hyrax's ability to transmit responses in multiple encodings.

EC2 Caching

We did not specifically explore the effect of EC2 caching (saving recently requested data on the EBS storage of the EC2 instance) on performance, but we do have a common-sense observation. The EBS volume utilized by the EC2 instance must be large enough to accommodate at least one of the largest granules for the service to work at all, and for caching to have much real value the EBS volume should be large enough to accommodate many granules.

A1 is the only architecture that in this study not only employs but relies on caching. In this study, a caching capability was not included in the proof of concept implementations of A2 and A3. In Hyrax, A2 and A3 can be implemented to utilize caching in the future. In A1 when a DAP request is received, the server determines if the granule that contains the requested data is already in the cache and only retrieves it from S3 if it is not. This clearly improves the performance of a series of requests against the same granule. These could be for different spatial/temporal subsets from the granule or for different variables from the same granule. We limited the A1 cache size so that only a single granule could ever be held. Even then if multiple DAP requests are received against the same granule, A1 will utilize the cached granule. For this reason, many of the tests are carefully designed to not make consecutive requests from the same granule to neutralize this caching effect in A1.

S3 Retrievals

In A1 the number of S3 requests required to generate a response is at most one, because of the caching mentioned above. The number of S3 requests required in A2 and A3 depends on the

details of the data storage and the relationship of the requests to that storage. For example, if a variable is stored contiguously, i.e. in one piece, then one request will always return the entire variable. If the variable is in multiple chunks, then the number of S3 requests depends on the relationship between the chunk layout and the requested area of interest. It will always be between one and the total number of chunks in the variable.

As stated previously, the proof-of-concept implementation of A2 and A3 do not utilize caching: Multiple requests against the same variable in a granule result in multiple, redundant S3 transfers.

Estimating Cost

Controlling costs incurred by using AWS Cloud services is very important for maintaining the value over similar on-premises IT resources. Because AWS Cloud costs are determined solely based on usage, active monitoring is a must regardless of the complexity of the system architecture implemented in the AWS Cloud. However, prior to implementation, a realistic cost model of the AWS Cloud services and how they integrate into the system should guide the design process to avoid unexpectedly high costs right from the start.

For this study, we relied on the AWS cost and usage reports to exactly find out which AWS Cloud services were used and their prices. Usage modelling (DAP requests) was based on Hyrax logs obtained from several NASA DAACs.

Cost and Usage Information

Billing reports provide information about the costs related to using AWS Cloud resources. The billing period is always one calendar month. Some types of billing reports can be provided during a billing period but that information represents only estimates of the true costs. The final bill is provided after the end of each billing period. Billing information can be presented in different ways:

- Hourly, daily, or monthly
- Per account for multi-account users
- AWS product or product resource
- User-defined tags

There are several types of billing reports: AWS Usage Report, Detailed Billing Report, and AWS Cost and Usage reports. Of those, only the AWS Usage Report is provided by default, as a high-level aggregation of costs over entire billing period.

For this study, we opted to use the Detailed Hourly AWS Cost and Usage Report with Resources and Tags. Reports of this type have the highest level of granularity both in time (per hour) and AWS resources. Each unique combination of AWS product, its resource type, and operation will produce a separate line in these reports for any fraction of an hour when used. For

example, the final January 2017 report has 15,814 entries of hourly cost and usage information, each described with up to 73 columns in the table.

OPeNDAP Logs

OPeNDAP logs are standard web server log files in the Apache HTTPd Common Log Format. Such files are produced by the web server that is part of the Hyrax server's front-end for handling incoming DAP requests. Producing these log files is a very common, practically required, configuration setting for all web servers and because of that there are many software tools that can parse such log entries.

The importance of OPeNDAP logs is that the information they contain about DAP requests is the best source of information for something like this study:

- Type of DAP request (metadata or data).
- Granule that contains requested data.
- Variables in the granule that are being requested.
- What kind of constraint expressions (data sub-setting), if any, is being utilized. This information is invaluable for more accurately estimating the number of S3 requests for A2 and A3
- Frequency of repeated access to same variables and granules.
- Amount of data returned as responses to DAP requests.

We have identified the information that should be available in these logs for maximum usefulness as the following:

- Date & Time (with time resolution of 1 second)
- Hashed Client internet protocol address (preferably with SHA2)
- Request (URL path + query string)
- Response Size
- Requester's User Agent

Internet Protocol addresses in the log entries can be anonymized to protect requester privacy if the anonymized values can still be unambiguously differentiated for their original values. Anonymizing the user-agent field (which may be something desired by a provider of the log files) renders the value of the user-agent field nearly worthless for analysis. The file can still be used to identify patterns of access using the user-agent hash, but it eliminates the possibility of linking the patterns to specific software. In addition, an anonymized user-agent field means that identifying which requests are being made by robots, crawlers, or directed programmatic data access must be done using inference.

Cost Model

Costs fall neatly into two general categories: fixed and dynamic. Fixed costs are generally applicable to all three architectures while dynamic costs are related to the specific architecture.

Examples of fixed costs are:

- Data storage in S3
- AWS outbound data resulting from running Use-Cases
- EC2 instance(s)

Examples of dynamic costs are:

- The number of S3 requests made by the Hyrax server.
- The data flow between the S3 bucket and the Hyrax server if they are not located in the same AWS region.

Not considered in the cost modeling are:

- EC2 instance type for the Hyrax server.
- How many Hyrax servers to run at any one time.
- Other types of data storage available in AWS, like Glacier or S3 Infrequent Access.
- Usage of any other AWS services that could make the proposed architectures more efficient or robust, like Lambda, Amazon Simple Notification Service, etc.

The common configuration for all three architectures were:

- One EC2 instance of the same type for the Hyrax server.
- Sample HDF5 data stored in S3.
- Both the EC2 instance and the S3 bucket with sample data are in the same AWS region.

In addition to the fixed costs of staging data in S3 and running Hyrax as an EC2 instance, all other usage costs stem from user DAP requests. Ability to model them appropriately is crucial for estimating realistic total costs. Properties of user DAP requests of interest for modeling costs are:

- type (metadata, data),
- if a data request:
    - sub-setting, or
    - aggregation, or
    - file download;
- source data granules(s).

Study Results

Architecture 1

Performance

The performance of A1 varies depending on two factors: complete granule size, and EC2 caching. The complete granule size affects the performance because A1 must copy the entire granule from S3 to EC2 before Hyrax can service the request. Figure 1 shows the S3 response times for A1. Each dot shows the total response time vs. the number of bytes sent for a single

granule transfer. The data are divided into three groups depending on granule size. The MERRA2 granules are in the two groups on the left as they have two sizes (~112 and ~193 MB). Their transfer times are mostly between 1.2 and 8 seconds with a few up to 20 seconds. The AIRS granules make up the group on the right as they are significantly larger (300-320 MB) and their transfer times are mostly in the range from 4 to 24 seconds. Two AIRS transfers that are not unusual in terms of granule size took well over a minute to complete. Through the testing, we have seen S3 operate such that most requests are relatively quick, 4 seconds in the AIRS example. But frequently and without an identifiable pattern an identical request took many times longer. This variation appears to be an S3 issue and is beyond the scope of this work to diagnose.

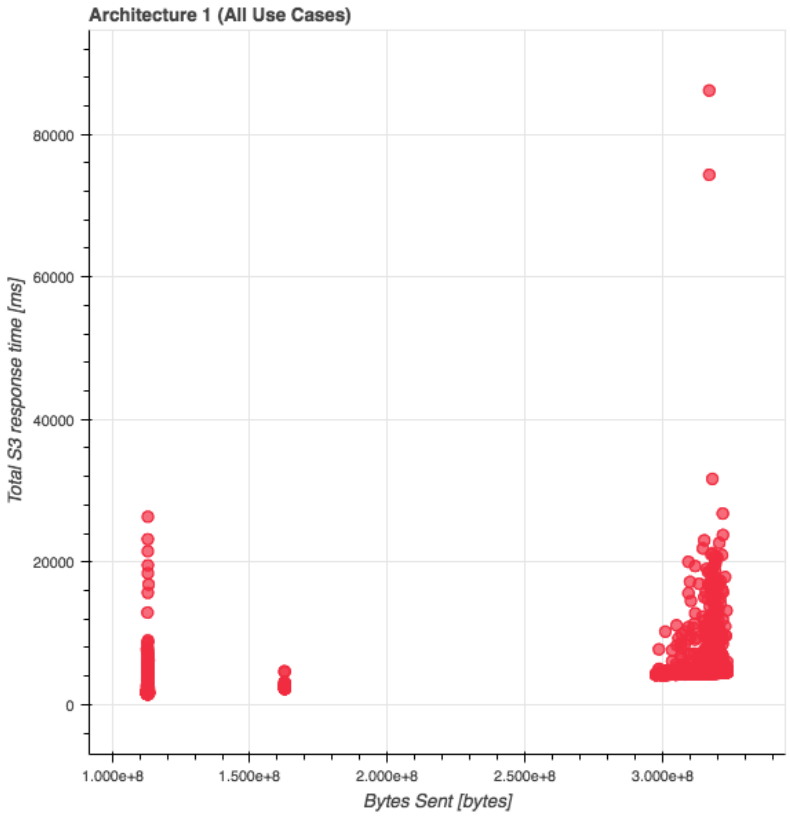

Figure 1: Architecture 1 (All Use Cases).

A1 caches granules on EC2 because it utilizes the regular Hyrax data handlers (the hdf4_handler in this study) to service the request. Once the granule has been copied to the EC2 instance, Hyrax works on it like any other granule. We sized the cache for our A1 instance so that it would only hold a single granule to minimize unwanted caching effects. Regardless, if the A1 code receives 3 consecutive DMR requests for the same HDF5 granule on S3, DMR output generation time drops significantly for the second and third requests due to cache effect (Figure 2). A1 data access is roughly twice as fast as data access with A2 and A3 when the requested HDF5 granule is resident in the Hyrax cache (on EC2) at the time the request is received. In general, we structured our tests to avoid making multiple consecutive requests for data from the same granule to eliminate this cache effect.

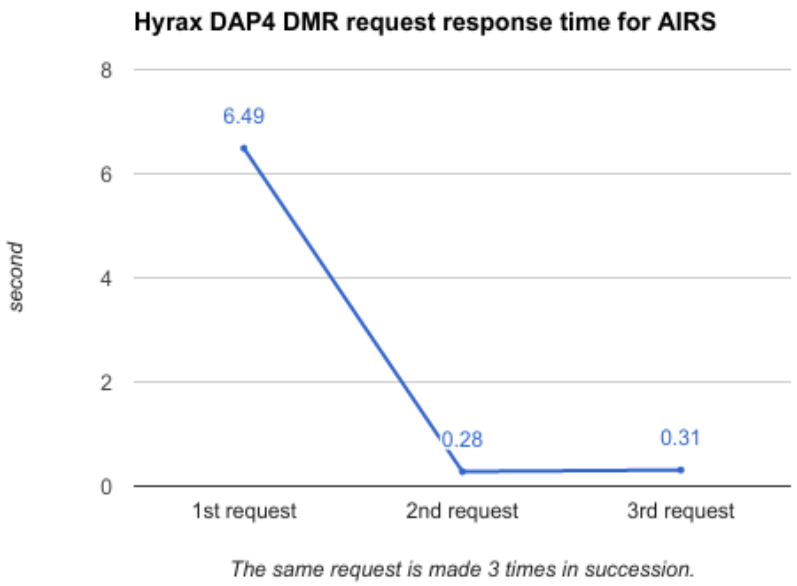

Figure 2: The effect of cached data on the performance of Architecture 1.

Cost

Table 8 shows the detailed cost data for A1 as reported in the hourly AWS cost and usage reports for one complete run of all the Use-Cases.

Table 8: Breakdown of reported AWS costs for Architecture 1.

AWS Service	Usage Type	Operation	Cost (US$)
AmazonEC2	BoxUsage:m4.xlarge	RunInstances	12.7
	DataTransfer-In-Bytes	RunInstances	0.00
	DataTransfer-Out-Bytes	RunInstances	0.00
	DataTransfer-Regional-Bytes	PublicIP-In	0.00
		PublicIP-Out	0.000872
	EBS:SnapshotUsage	CreateSnapshot	0.065
	EBS:VolumeUsage.gp2	CreateVolume-Gp2	0.821
	EBSOptimized:m4.xlarge	Hourly	0.00
AmazonS3	DataTransfer-Out-Bytes	GetObject	0.00
	Requests-Tier1	ListBucketVersions	0.0275
		PutObject	0.00005
	Requests-Tier2	GetObject	0.00788
		HeadBucket	0.000258
		ReadACL	0.0000016
	TimedStorage-ByteHrs	StandardStorage	0.095
	USE1-USW2-AWS-Out-Bytes	GetObject	0.000272
		HeadBucket	0.00000522
		ListBucketVersions	0.0422
awskms	us-east-1-KMS-Requests	ListAliases	0.00

The total cost was $13.75, as reported in 437 entries over a 19-hour-long period. The largest cost comes from running the EC2 instance for the Hyrax server (92.29%) and its EBS volume (5.98%). Since these are the most expensive AWS services, the longer run time of the Use-Cases is the direct driver for these costs. The cost of S3 data storage is only 0.69% of the total cost, while the cost of Hyrax's S3 requests for HDF5 granules is at 0.06%.

Architecture 2

Performance

There are two major functional differences between A1 and A2:

1. The number of bytes transferred between S3 and the Hyrax server on EC2. In A2 (and A3), the transfer size is the sum of sizes of the chunks required to fulfill the data request. A2 data requests returned between 217 and 831,744 bytes compared to an average granule size of 210,308,700 bytes in A1.
2. The number of S3 requests made by Hyrax to retrieve the data from S3. In A1 Hyrax makes a single HTTP GET request to retrieve the granule. In A2 retrieving 1 MERRA2 variable composed of 16 chunks causes 16 GET requests to be issued to S3. Requesting all of the variables in a MERRA2 granule from A2 (or A3) results in the server issuing 3107 GET requests to S3. Similarly, for the AIRS data, although the AIRS variables we examined have 8 chunks.

The performance of A2 is also much better than A1 when small single variable data requests involve subsets across many granules. For example, in use-case 6 where we subset one data point from each granule and aggregate each point over time from hundreds of granules, A2 performs 4 to 30 times faster than A1 (Figure 3).
This performance advantage disappears quickly as the number of requested variables increases. The MERRA2 granule contains 197 variables. Asking for 16 of them caused A2 to take twice as long as A1 to service the request and A3 took 3 times longer. Doing the same with 30 AIRS variables (of the 777 in the granule) caused A2 to respond in 1.7 times the time of A1 and A3 responded in 2.8 times the time of A1.

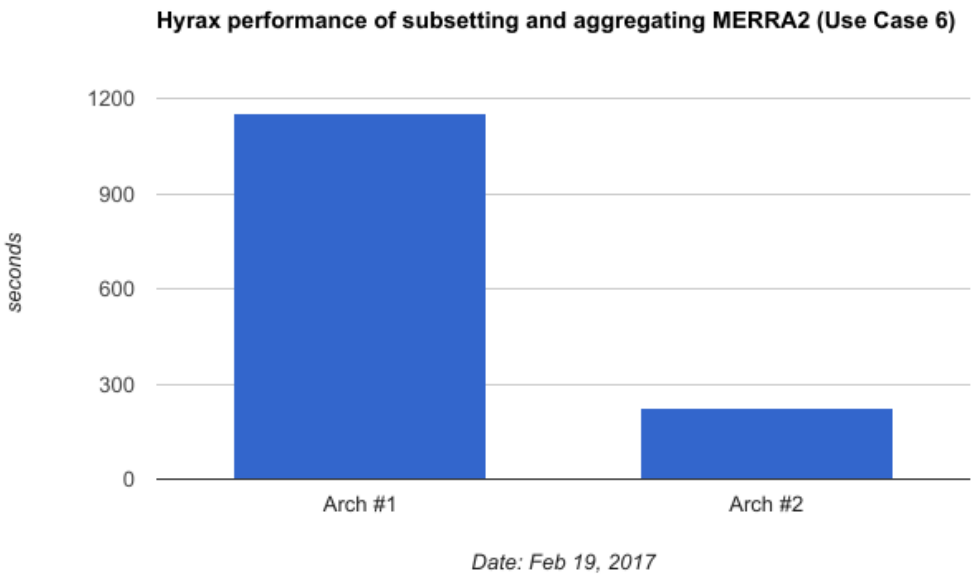

Figure 3: The cumulative effect of performance gain: transferring whole HDF5 granule vs. HDF5 chunks per data subset request

The size of the data transfers in A2 and A3 is small enough (<1MB) that the performance is impacted by S3 start-up time rather than the transfer time. Figure 4 shows that for most cases, the S3 turnaround time is > 90% of the total time.

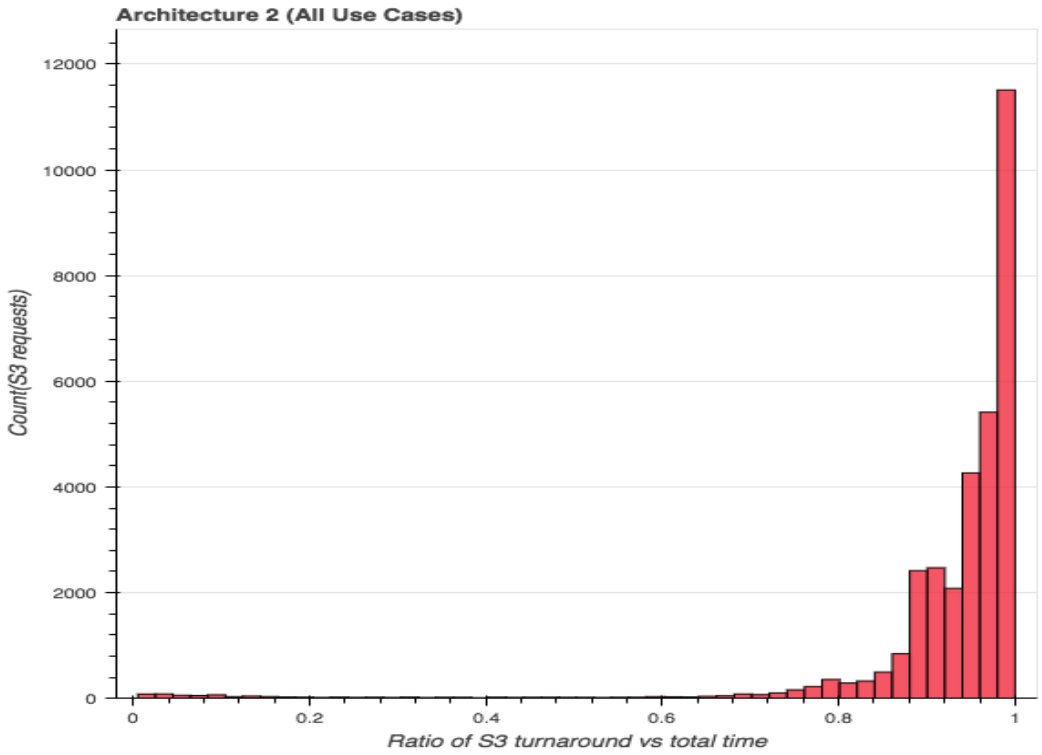

Figure 4: Architecture 2 (All Use Cases)

Cost

Table 9 holds the cost data as reported in the detailed hourly AWS cost and usage reports for one complete run of all the Use-Cases.

Table 9: Breakdown of reported AWS costs for Architecture 2 and all Use-Cases.

AWS Service	Usage Type	Operation	Cost (US$)
AmazonEC2	BoxUsage:m4.xlarge	RunInstances	1.72
	DataTransfer-In-Bytes	RunInstances	0.00
	DataTransfer-Out-Bytes	RunInstances	0.00
	DataTransfer-Regional-Bytes	PublicIP-In	0.00
		PublicIP-Out	0.00133
AmazonS3	Requests-Tier1	ListBucketVersions	0.00257
	Requests-Tier2	GetObject	0.0149
		HeadBucket	0.0000248
	TimedStorage-ByteHrs	StandardStorage	0.0101
	USE1-USW2-AWS-Out-Bytes	GetObject	0.000134
		HeadBucket	0.00000051
		ListBucketVersions	0.00395

The total cost was $1.75, as reported in 39 entries over a two-hour-long period. The largest cost comes from running the EC2 instance for the Hyrax server at 98.12% of the total. The small total cost for this architecture is largely due to only requiring two billable hours to complete all the Use-Cases. The cost of S3 data storage is only 0.46% of the total cost, while the cost of Hyrax's S3 requests for HDF5 granules is at 0.79%.

Architecture 3

Performance

The performance of A3 is similar to the performance A2. When the request only accesses a single chunk of a single variable in each granule, such as in Use-Case 6 which retrieves a single pixel from each granule, A3 performs slightly better than A2. If several chunks are involved, such as in Use-Case 11 which retrieves an entire variable, it performs slightly worse than A2 (Figure 5). A3 suffers, as A2 does, when the number requested variables increase, and in fact A3 is consistently the worst performing of the three architectures for requests involving many variables (Use-Case 18 and 19) and when all the variables are requested (Use-Case 20 and 21).

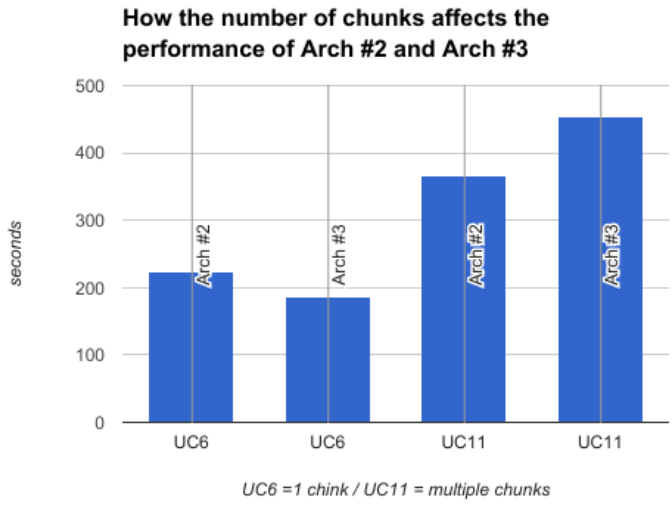

Figure 5: Subset and aggregation performance of Architecture 2 and Architecture 3 depends on the number of chunks involved.

Cost

Table 10 holds the cost data as reported in the detailed hourly AWS cost and usage reports for one complete run of all the Use-Cases.

Table 10: Breakdown of reported AWS costs for Architecture 3 and all Use-Cases.

AWS Service	Usage Type	Operation	Cost (US$)
AmazonEC2	BoxUsage:m4.xlarge	RunInstances	1.72
	DataTransfer-In-Bytes	RunInstances	0.00
	DataTransfer-Out-Bytes	RunInstances	0.00
	DataTransfer-Regional-Bytes	PublicIP-In	0.00133
		PublicIP-Out	0.00133
	USE1-USW2-AWS-In-Bytes	PublicIP-In	0.00
	USE1-USW2-AWS-Out-Bytes	PublicIP-Out	0.00
AmazonS3	Requests-Tier1	ListBucketVersions	0.00327
	Requests-Tier2	GetObject	0.0138
		HeadBucket	0.0000236
	TimedStorage-ByteHrs	StandardStorage	0.00806
	USE1-USW2-AWS-Out-Bytes	GetObject	0.000124
		HeadBucket	0.00000048
		ListBucketVersions	0.00502

The total cost was $1.75, as reported in 43 entries over a two-hour-long period. The largest cost comes from running the EC2 instance for the Hyrax server at 98.12% of the total. The small total cost for this architecture is largely due to only requiring two billable hours to complete all the Use-Cases. The cost of S3 data storage is now 0.46% of the total cost, while the cost of Hyrax's S3 requests for HDF5 granules is at 0.79%.

Appendix 1. Overview of System Components

Background

Thanks to the range-get support in HTTP/1.1, it is possible to subset HDF5 data using typical web servers such as Apache Tomcat or HTTPD without relying on Hyrax. The caveat is that the HTTP client should know precisely what ranges of bytes should be requested and how to correctly interpret the chunk returned from the HDF5 granule on web server. Such operations are typically done by HDF5 library internally but HDF5 library operation is limited to POSIX-based local files only.

DAP Protocol

The OPeNDAP Data Access Protocol (DAP) is a data transmission protocol designed specifically for science data. The protocol relies on the widely used and stable HTTP and Multipurpose Internet Mail Extensions (MIME) standards, and provides data types to accommodate gridded data, relational data, and time series, as well as allowing users to define their own data types. The DAP has been in use since 1995 and is a NASA Community Standard.

Hyrax Server

Hyrax is OPeNDAP Inc.'s implementation of a DAP server. Hyrax is two-tiered system with a public facing front end application (the OPeNDAP Lightweight Frontend Service, OLFS) and a private facing back end application (the Back End Server, BES) that handles all of the data access and heavy lifting for the server. The OLFS is implemented as a Java servlet which is typically deployed in Apache Tomcat. The OLFS looks at each request and formulates a query to the BES which handles reading data from the data stores and returning DAP-compliant responses to the OLFS. In turn, the OLFS may pass these responses back to the requestor with little or no modification or it may use them to build more complex responses. The nature of the inter-process communication (IPC) between the OLFS and BES is such that they should both be on the same computer or be able to communicate over a very high bandwidth channel.

HDF5 Library

HDF5 is a data model, several code libraries, and file format for storing and managing data. It supports an unlimited variety of data types, and is designed for flexible and efficient input and output of high volume and complex data. HDF5 is portable and extensible, allowing applications to evolve in their use of HDF5.

HDF5 has found widespread use in many scientific and engineering applications that deal with multidimensional numerical arrays. It is a de facto standard for storing data in Earth Science, synchrotron and particle physics, and high performance computing centers around the world.

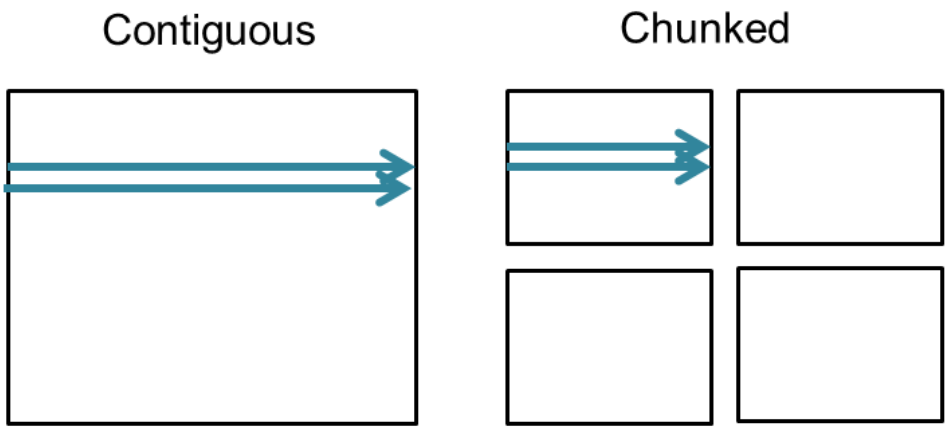

Figure A1-1: Diagram of contiguous and chunked variable layout. Contiguous storage holds multiple array rows in a contiguous region of the disc. Chunked storage breaks arrays into multi-dimensions blocks, each stored contiguously.

The most important elements of the HDF5 data model are: Groups, Datasets, and Attributes. (Note: In this document, we use the term variable to refer to the HDF Dataset) Groups establish hierarchical relationships among other HDF5 objects much like folders/directories in a file system. Datasets are the main data payload object. They typically represent multidimensional arrays of elements of the same data type. Attributes also carry data and have the same size limitations as Datasets. However, their role is to hold data that provide the context (self-description) for the data in the Groups and Datasets.

One feature of HDF5 variables that is important to this study is their storage layout. There are three different ways that the elements belonging to one HDF5 variable can be stored in an HDF5 granule file:

1. contiguous – all variable elements are physically adjacent to each other in one block;
2. chunked – variable elements are stored throughout the granule file in equal-sized chunks of a pre-defined size (as illustrated by Figure A1-1);
3. compact – variable elements are stored contiguously in the variable's object header.

The compact storage arrangement is applicable only to HDF5 variables smaller than 64 kilobytes and is therefore seldom used.

Chunked storage is required for HDF5 variables which can have their size altered with new data (extendable) or which have one or more unlimited dimensions. Using this layout can greatly improve performance for large variables because only the chunks with data of interest need to be read or written. However, selecting the chunk size without considering read/write access patterns may also cause seriously poor performance.

With a chunking layout, it is possible to process each chunk's data on their way to the granule file (write operation) and apply the reverse processing on the same chunk's data on their way from the granule file (read operation). This is the task for HDF5 filters and the HDF5 library has several built-in. It is possible also to supply third-party filters. Typically, filters are used to reduce the size of the stored data in the granule file (compression), or to scale the values according to

a formula. The HDF5 library automatically handles invocation of selected filters in correct order for both read and write operations.

Amazon Web Services (AWS)

Amazon Web Services (AWS) is a subsidiary of Amazon.com launched in 2006. AWS offers an ever-growing suite of Internet-accessible services that enable utilization of on-demand computing systems. These systems are commonly referred to as *cloud computing*. Cloud computing enables access to various IT resources (compute power, storage, database, etc.) with the cost based entirely on usage (pay-as-you-go pricing). This new approach to procuring IT resources was largely enabled by the advances in *virtualization* (ability to host several virtual computers independently on a single physical computer hardware), continuous decline in data storage hardware cost, and general availability of faster networking and Internet access.

Architecting computer systems based on cloud computing can help lower the cost of providing and managing comparable on-premises computing resources, and dramatically improve responsiveness to changing processing demands. By aggregating usage from many customers, cloud providers can achieve higher economies of scale on their IT resources than any individual customer. The savings are further possible by utilizing cloud services only when required, thus scaling cloud-computing resources in sync with the need.

AWS is available from sixteen global Regions. The Regions are physical locations distributed globally in proximity to major IT markets. Each Region has several Availability Zones (AZ), 42 in total. Availability Zones consists of one or more discrete data centers with physical computing hardware and supported by redundant power, networking, and connectivity.

AWS currently offers a wide range of services from basic (compute, storage, hosted relational databases) to very sophisticated, e.g. preconfigured computing frameworks for analytics (Spark, MapReduce), artificial intelligence/machine learning, or Internet-of-Things (IoT).

What follows are brief introductions of the AWS services relevant for this report.

Amazon Elastic Compute Cloud (EC2)

Amazon EC2 is a service for providing compute resources in AWS. These computing resources are in the form of virtual computers (*EC2 instances*) available in a variety of operating systems, type and number of CPUs, and memory capacity. It is also possible to create customized templates, called *Amazon Machine Images* (AMI), and procure EC2 instances based on them to have a number of servers running with pre-installed software and data in the exactly the same manner.

There are many EC2 instance categories, optimized for a range of specific tasks: high performance computing (HPC) clusters, high data input/output, massive floating-point processing capacity (graphics processing unit (GPU) Compute and GPU Graphics instances), etc.

The OPeNDAP Hyrax servers for this study were run as EC2 instances.

Amazon Elastic Block Store (EBS)

Amazon EBS furnishes persistent block storage volumes for EC2 instances in the AWS Cloud. One EBS volume can belong only to one EC2 instance. Although some higher end EC2 instances do come with local block volumes, called *instance storage*, EBS volumes are the only way to provide file system-like data storage for most of the default EC2 instances. It is also one option for preserving any data after EC2 instances are shut down. EBS volumes can have their size changed on-the-fly and support point-in-time backups. These EBS snapshot backups can be used by other EC2 instances.

EBS volumes come in two general types: solid state drive (SSD) and hard disk drive (HDD), and each offer a few subtypes optimized for throughput or general use. Every EBS volume is automatically replicated within its Availability Zone for increased reliability and data protection.

Amazon Elastic File System (EFS)

Amazon EFS is a service for providing data storage for EC2 instances with a standard file system interface (POSIX). EFS's functionality is akin to network-shared file storage (NFS) in on-premises computer systems. Storage capacity scales automatically so the charges are related to only capacity that is actually being used at hourly sampling. EFS storage is shareable among EC2 instances and is one approach for establishing common data sharing for applications that may run on multiple EC2 instances.

It is also possible to connect Amazon EFS with on-premises servers as a mean of migrating data or backing up data to the AWS Cloud.

Amazon Simple Storage Service (S3)

Amazon S3 provides *object storage* service for the AWS Cloud. This storage architecture differs from Amazon EBS and EFS as data are managed as objects associated with unique identifiers within a collection. Amazon S3 refers to such collections of data objects as *buckets*. There are no limits for the number of objects in one bucket, and each object can be up to 5 terabytes in size.

In traditional file systems, data is stored in files within a hierarchy of folders/directories. Object storage on the other hand has no hierarchy; each object is accessed by its unique identifier, known as *the key*. Thus, object storage can be viewed as an application of associative arrays to data storage.

Amazon S3 implements a Representational state transfer (RESTful) Application Programmer Interface (API) for object access, which is another difference from the file system interfaces. This allows it to be used as a serverless data web storage, enabling any application that "speaks" HTTP protocol to directly and easily access its objects.

Amazon S3 is designed to be massively scalable and with high level of data durability (stated: 99.999999999%). Its price is aimed at making it an AWS storage service that is economical for storing large amounts of data and accessing them efficiently whenever required. There is also a lower-priced service tier called *S3 Standard - Infrequent Access* aimed at storing infrequently accessed data. S3 buckets can be configured to automatically migrate objects to the S3 Infrequent Access storage.

Appendix 2: Architecture Details

Architecture 1: Baseline

The most straightforward way to serve HDF5 data from S3 using Hyrax is simply to use S3 as a granule storage device and copy entire granules required to service a data request from S3 to the EC2 instance that receives the data request (see Figure A2-1). Once the granule is transferred, Hyrax serves HDF5 data as usual from local storage using the Hyrax's HDF5 handler. This architecture also includes an XML index file, similar to a THREDDS catalog, that maps the physical data storage to a user interface and includes some access and service related metadata. In addition, retrieved granules are cached on the EC2 instance's local storage (EBS Cache in Figure A2-1) and accessed without redundant transfers from S3 to EC2. OPeNDAP initially developed this baseline architecture several years ago with support from the National Oceanic and Atmospheric Administration (NOAA). We adopted it as a baseline for this study because the code was available and functional and it allowed us to establish a baseline quickly.

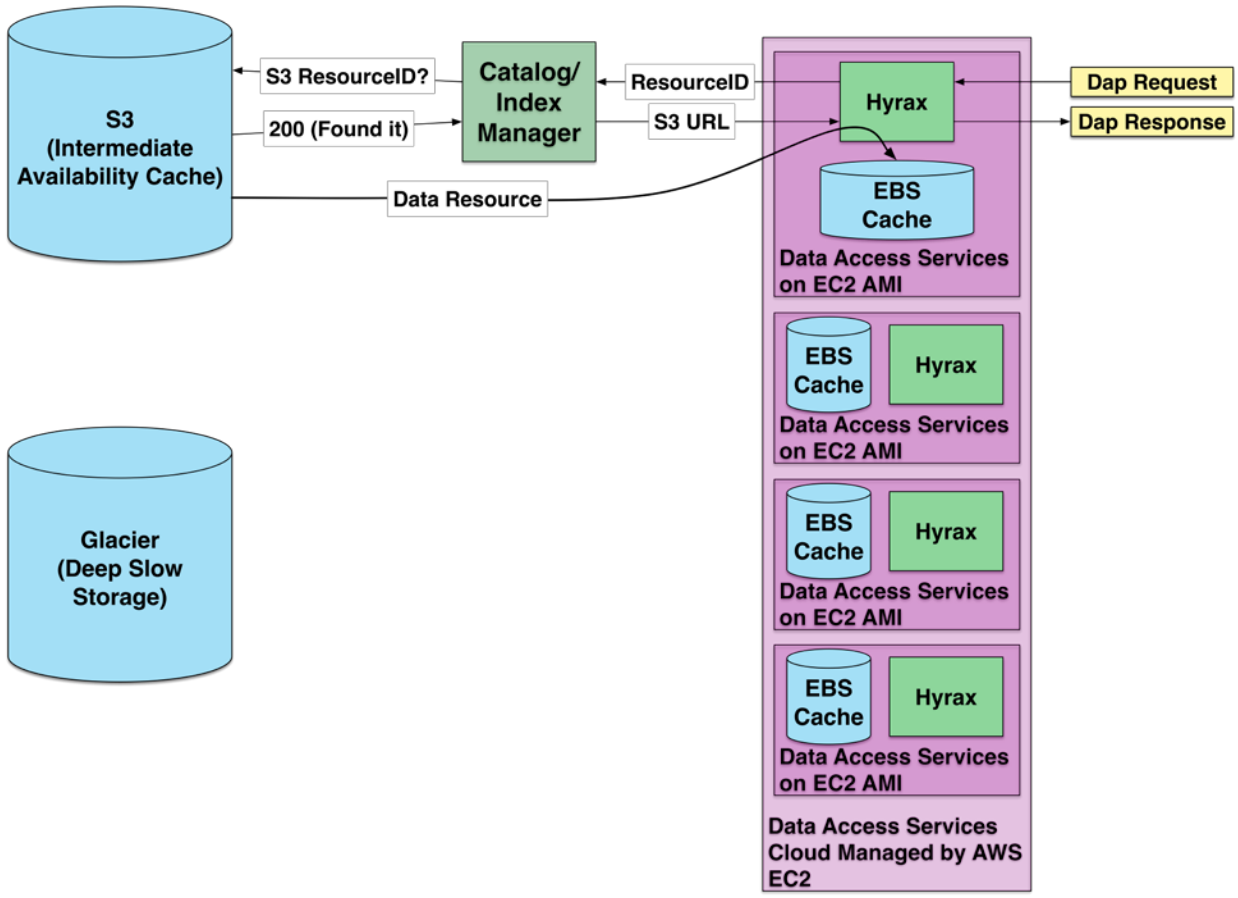

Figure A2-1: Diagram of Architecture 1 in which complete files are stored in Amazon S3 and retrieved as a single object. Once retrieved, multiple OPeNDAP requests for data in each file can be serviced without copying the file again.

Architecture 2: Files with HTTP Range GETs

A2 is identical to A1 in terms of data storage, i.e. entire granules are stored as objects, but differs in the access and retrieval approach. An important capability of Hyrax allows users to request subsets of granules using constraint expressions in the query string of OPeNDAP URLs. When Hyrax is serving data from the local filesystem the HDF handler translates the user request into the minimal series of file seeks and reads that are required to satisfy the request. This decreases the amount of unnecessary data retrieved and transferred to the user and simplifies many data management tasks on the user side.

The S3 HTTP interface allows the retrieval of partial objects data using the HTTP/1.1 Accept-Ranges request. In this case, the request includes the size and offset of the desired data along with an object identifier and only that data is retrieved and transferred from S3 to EC2.

A2 takes advantage of the Range GET capability (see Figure A2-2) by translating the DAP request into a series of range-gets required to fulfill the request. This approach minimizes the amount of data transferred between S3 and EC2 as Hyrax does not have to transfer the entire HDF5 from S3 as shown in A1.

The maximum number of S3 requests that the server will make is the number of chunks in each of the requested variables. For contiguous storage layout case, it will be 1. If some chunks are consecutive (e.g., chunk 1 ends at 10 and chunk 2 begins at 11), the number of requests can be reduced when data request involved consecutive chunks.

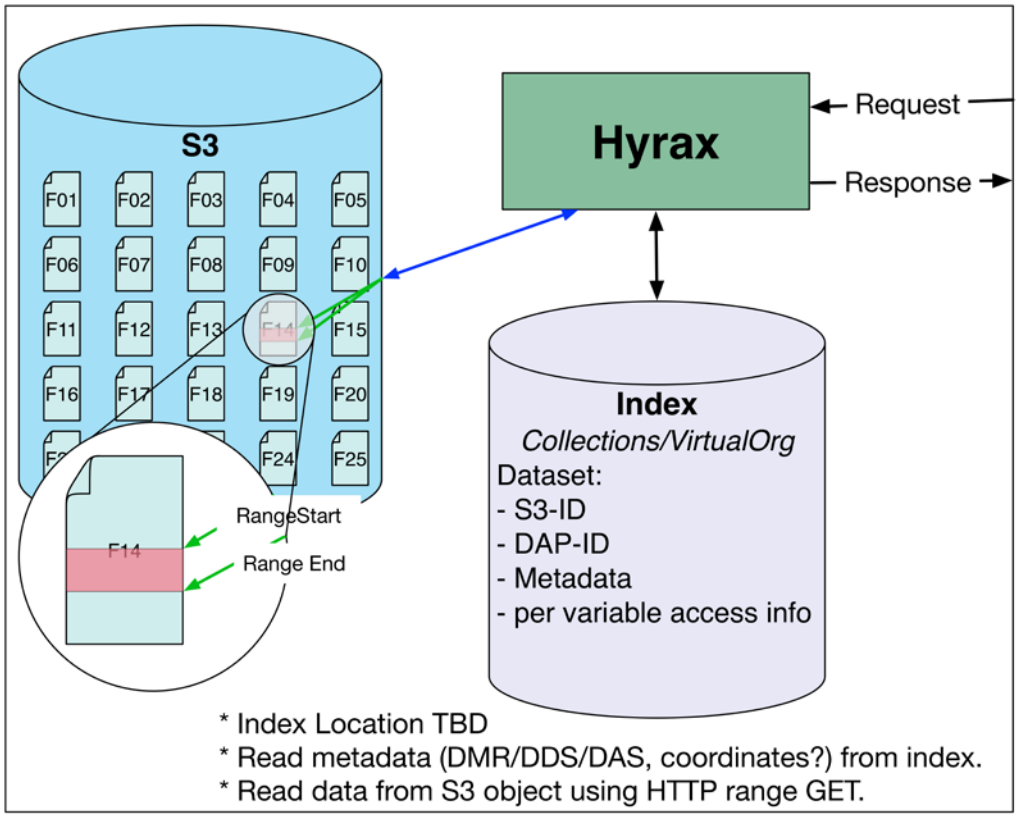

Figure A2-2: Diagram of Architecture 2 in which complete files are stored in S3 and OPeNDAP requests against those files are serviced by retrieving blocks of bytes from a known location in the file. These retrievals are termed range-gets. Their locations are stored in an external index.

Architecture 3: Shredded HDF5 Files

The third architecture we explored is closest to the actual object store concept in that we “shred” the original granule files into individual objects that are the contiguous variables or chunks in the original granule (see Figure A2-3). The md5 checksums added to the DMR representation (described above) are used as unique identifiers (keys) for each object. In this case, the Hyrax server access a series of objects required to service a user request in exactly the same way as they are accessed in a granule. Instead of utilizing a filesystem to seek to an offset position in a file and read the bytes in order to retrieve data, the software makes an HTTP request for the same data using the md5 checksum recorded in the DMR++ file.

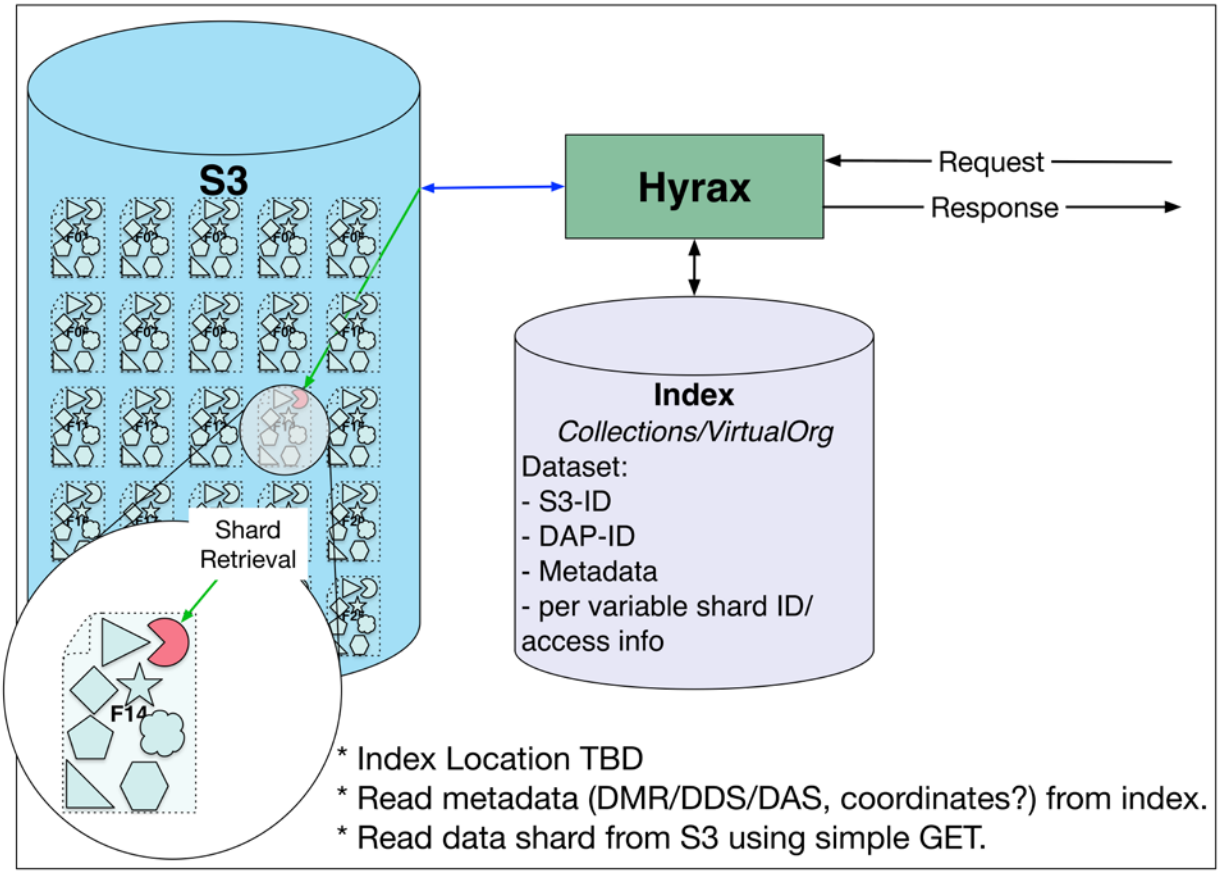

Figure A2-3: Diagram of Architecture 3 in which files are split into multiple pieces (shards) and those shards are stored in S3. OPeNDAP requests are serviced by copying requested shards directly from S3. In this case, the original data are preserved as shards, but the original files are not preserved.

Appendix 3. Use Case Description

Use-Case #	Tag	Description
1	n/a	Access the **CF-enabled** DAP2 Hyrax Dataset Descriptor Structure (DDS), Dataset Attribute Structure (DAS) and Data responses (.das, .dds, .dods).
2	UC2_A1CFT, UC2_A2CFT, UC2_A3CFT	Access the **CF-enabled** DAP4 Hyrax DMR and Data responses (.dmr, .dap).
3	n/a	Access the **default** DAP2 Hyrax DDS, DAS and Data responses (.das, .dds, .dods).
4	n/a	Access the **default** DAP4 Hyrax DMR and Data responses (.dmr, .dap).
5	n/a	Get netCDF-4 files with the CF-enabled DAP2 Hyrax FileOut netCDF responses (.nc4).
6	UC6_A1CFT, UC6_A2CFT, UC6_A3CFT	This test emulates a subset request for the MERRA2 aggregation in which a single value is retrieved from every aggregation member granule. Nominal client query: PRECCU[0:1:0][180:1:180][288:1:288]
7	UC7_A1CFT, UC7_A2CFT, UC7_A3CFT	This test emulates a subset request for the AIRS aggregation in which a single value is retrieved from every aggregation member granule. Nominal client query: Temperature_A[0][0][0]
8	n/a	Access the **CF-enabled** DAP2 Hyrax DDS, DAS and Data responses of an HDF5 granule with contiguous storage (.das, .dds, .dods).
9	n/a	Access the **CF-enabled** DAP4 Hyrax DMR and Data responses of an HDF5 granule with contiguous storage.
10	UC10_A1CFT, UC10_A2CFT, UC10_A3CFT	This test emulates a subset request for the AIRS aggregation in which a 3x6x21 pixel area is retrieved from every aggregation member granule. In this test the location of the area was chosen such that **2 of the 8 chunks** of the variable are required to formulate the response. Nominal client query: Temperature_A[13:1:15][40:1:45][175:1:195]
11	UC11_A1CFT, UC11_A2CFT, UC11_A3CFT	This test emulates a subset request for the AIRS aggregation in which the requested variable is decimated for every aggregation member granule. Nominal client query: Temperature_A[0:8:23][0:15:179][0:15:359]
12	UC12_A1CFT, UC12_A2CFT, UC12_A3CFT	This test emulates a subset request for the MERRA2 aggregation in which a 41x51 pixel area is retrieved from every aggregation member granule. In this test the location of the area was chosen such that **4 of the 16 chunks** of the variable are required to formulate the response. Nominal client query: PRECCU[0:1:0][160:1:200][245:1:295]
13	UC13_A1CFT, UC13_A2CFT, UC13_A3CFT	This test emulates a subset request for the MERRA aggregation in which the requested variable is decimated for every aggregation member granule. Nominal client query: PRECCU[0:1:0][0:60:360][0:8:575]

14	UC14_A1CFT, UC14_A2CFT, UC14_A3CFT	This test emulates a subset request for the MERRA2 aggregation in which a 40x50 pixel area chosen at *random* from the target array and is then retrieved from every aggregation member granule. This is repeated *50 times* with a new subset area chosen each time. Nominal client query: PRECCU[0][160:1:199][245:1:294]
15	UC15_A1CFT, UC15_A2CFT, UC15_A3CFT	This test emulates a subset request for the AIRS aggregation in which a 40x50 pixel area chosen at *random* from the target array and is then retrieved from every aggregation member granule. This is repeated *50 times* with a new subset area chosen each time. Nominal client query: Temperature_A[0:8:23][0:1:39][0:1:49]
16	UC16_A1CFT, UC16_A2CFT, UC16_A3CFT	In this test a 40x50 pixel subset request for the MERRA2 aggregation is placed at a *random* location the target array and retrieved from the granule file. Each granulein the list is accessed with a different subset location. Nominal client query: PRECCU[0][160:1:199][245:1:294]
17	UC17_A1CFT, UC17_A2CFT, UC17_A3CFT	In this test a 40x50 pixel subset request for the AIRS aggregation is placed at a *random* location the target array and retrieved from the granule file. Each granule in the list is accessed with a different subset location. Nominal client query: Temperature_A[0:8:23][0:1:39][0:1:49]
18	UC18_A1CFT, UC18_A2CFT, UC18_A3CFT	This test requests 17 decimated variables from 100 MERRA2 granules used in the aggregation tests. Nominal variable subset looks like: PRECCU[0][0:60:360][0:8:575]
19	UC19_A1CFT, UC19_A2CFT, UC19_A3CFT	This test requests 30 decimated variables from 100 of the AIRS granules used in the aggregation tests. Nominal variable subset looks like: Temperature_A[0:8:23][0:15:179][0:15:359]
20	UC20_A1CFT, UC20_A2CFT, UC20_A3CFT	This test requests every variable (197 in all) from 100 of the MERRA2 granules used in the aggregation tests. There is no query string submitted with this test.
21	UC21_A1CFT, UC21_A2CFT, UC21_A3CFT	This test requests every variable (777 in all) from 100 of the AIRS granules used in the aggregation tests. There is no query string submitted with this test.

References

Globus. (2017, 03 07). *Research Data Management Simplified*. Retrieved from Globus: https://www.globus.org/

NASA. (2017, 3 9). *AIRS3STD: AIRS/Aqua L3 Daily Standard Physical Retrieval (AIRS-only) 1 degree x 1 degree V006*. Retrieved from NASA GES DISC: https://disc.sci.gsfc.nasa.gov/uui/datasets/AIRS3STD_V006/summary?keywords=%22AIRS%22

NOAA. (2017, 03 07). *Access Data*. Retrieved from NOAA: https://www.nodc.noaa.gov/access/services.html

OPeNDAP, Inc. (2007, 03 07). *The Data Access Protocol: DAP Version 4.0*. Retrieved from OPeNDAP Documentation: http://docs.opendap.org/index.php/DAP4:_Specification_Volume_1#Characterization_of_a_Data_Source

The HDF Group. (2017, 03 07). *HDF4 Mapping Project*. Retrieved from The HDF Group Support Website: https://support.hdfgroup.org/projects/h4map/

Unidata. (2017, 03 07). *Unidata | THREDDS Data Server*. Retrieved from Unidata: http://www.unidata.ucar.edu/software/thredds/current/tds/

Glossary

AWS	Amazon Web Services are reliable, scalable, and inexpensive cloud computing services that allow users to pay for what they use.
Deflate	A lossless data compression algorithm that uses a combination of the LZ77 algorithm and Huffman coding
DMR	Dataset Metadata Response is an XML representation of OPeNDAP resource.
DMR++	Dataset Metadata Response with additional offset/length information for each variable.
EBS	The Amazon Elastic Block Store is the storage file system used by an Amazon EC2 instance. EBS is one of the more expensive AWS storage systems
EC2	Amazon Elastic Compute Cloud is a web service that provides secure, resizable compute capacity in the cloud that allows users to run their applications like Hyrax server.
Hyrax	The next generation server from OPeNDAP
Kibana	A web-based analysis platform typically used with ElasticSearch (https://www.elastic.co/products/kibana)
LZ77	Lossless data compression algorithm published in a paper by Abraham Lempel and Jacob Ziv in 1977
MD5	Message Digest 5 is a hash function algorithm that produces 128-bit hash value.
NcML	NetCDF Markup Language is an XML representation of netCDF metadata.
netCDF-4	Network Common Data Form version 4 is a file format based on HDF5.
NSIDC	National Snow and Ice Data Center – A NASA DAAC for cryospheric data.
S3	The AWS Simple Storage Service is an object store that utilizes an HTTP interface to add and retrieve digital objects from a user defined “bucket”. S3 is one of the cheapest AWS storage system and trades performance for cost (like most AWS services).
SHA2	The Secure Hash Algorithm 2. It is a set of cryptographic hash functions designed by the National Security Agency (NSA). Cryptographic hash functions are mathematical operations run on digital data; by comparing the computed "hash" (the output from execution of the algorithm) to a known and expected hash value, a person can determine the data's integrity.
Shuffle	An HDF5 library filter that exploits the fact that, for many array variables, most of the entropy occurs in only a few byes of the elements’ values.

Acronyms

AIRS	Atmospheric InfaRed Sounder
API	Application Programmer Interface
AWS	Amazon Web Services
BES	Back End Server
BESCMDLN	Back End Server Command Line
CF	Climate Forecast [Convention]
DAAC	Distributed Active Archive Center
DAP	Data Access Protocol
DAS	Dataset Attribute Structure
DDS	Dataset Descriptor Structure
DMR	Dataset Metadata Response
EBS	Elastic Block Store
EC2	Elastic Compute Cloud
EFS	Elastic File System
GES DISC	Goddard Earth Sciences Data and Information Services Center
GB	Gigabyte
GPU	Graphics processing unit
HDF	Hierarchical Data Format
HTTP	HyperText Transfer Protocol
IoT	Internet-of-Things
IP	Internet Protocol [Address]
IPC	Inter-process communication
LaRC ASDC	Atmospheric Science Data Center at NASA Langley Research Center
LP DAAC	Land Processes Distributed Active Archive Center
MD5	Message Digest 5
MERRA2	Modern-Era Retrospective analysis for Research and Applications Version 2
MIME	Multipurpose Internet Mail Extensions
NcML	NetCDF Markup Language
netCDF-4	Network Common Data Form version 4
NASA	National Aeronautics and Space Administration
NOAA	National Oceanic and Atmospheric Administration
NSIDC	National Snow and Ice Data Center

OLFS	OPeNDAP Lightweight Frontend Service
OPeNDAP	The Open-source Project for a Network Data Access Protocol
RESTful	Representational state transfer
S3	Simple Storage Service
SHA2	Secure Hash Algorithm 2
THG	The HDF Group
THREDDS	Thematic Real-time Environmental Distributed Data Services
URL	Universal Resource Locator
UUID	Universally Unique Identifier
XML	eXtensible Markup Language