Objectstore Data handling within AusSRC

1 Background to AusSRC data-handling

1.1 Introduction

AusSRC provides software pipelines for the production of science-ready data products from Level 5 radio astronomy datasets. As such, one of its primary design goals is the efficient ingest of these datasets for processing by the various algorithms within these pipelines.

Traditionally, most of these (legacy) algorithms assume that their input data resides in an underlying POSIX filesystem, requiring the standard methods of file open, reading and writing to ingest the data. Typically, these algorithms will read in a few tens of Gb of data at a time, normally retrieved from a single “datacube” that may be one to several terrabytes in size.

The datacubes themselves are stored within archives hosted by supercomputing or cloud-computing facilities, which are now more often than not storing these large objects in “objectstores”, rather than traditional file systems. This is the case for the CSIRO ASKAP Science Data Archive (CASDA), hosted at the Pawsey Supercomputing centre, and will be the case for data from the upcoming SKA observatory archives. The Pawsey (production-ready) objectstore is ***Acacia***, available at https://projects.pawsey.org.au.

Here we look at various methods to ingest the objectstore data directly into the pipeline, rather than copying the data to a POSIX file system (“staging”) for later processing.

1.2 Note on uploading large datafiles to Objectstores

There are several clients available for uploading data to an objectstore; the Acacia help pages recommend using MinIO, a popular command-line client with unix-like cp, rm and ls commands. However, there is an issue when uploading very large files.

The maximum size for an atomic piece of data in an objectsore is 5Gb. Most objectstores get around this limitation by storing a large file in "chunks"; each chunk is under 5Gb in size and a manifest stored alongside the chunks details the relationships between the chunks. However, there is also a limit to how large this manifest file can be! Clients such as Minio use a rather small default chunk-size of 8.8Kb, so to upload an 800Gb file will take approximately 910 000 chunks. This results in a manifest file that is too large to store and the objectstore will unhelpfully return an "unexpected EOF" error, and purge all the chunks. This can be quite frustrating, as it can takeup to an hour or so to reach this point.

The solution is to change the chunk-size. With MinIO, this must be done in the MinIO configuration file, and will become the chunk-size for all further uploads. APIs such as python's Boto3 library can specify the chunk-size on each upload operation, eg:

```
import boto3
from boto3.s3.transfer import TransferConfig
client = boto3.session.Session().client(service_name='s3',
aws_access_key_id=******, aws_secret_key=******,
endpoint_url='https://projects.pawsey.org.au')
config = TransferConfig(multipart_threshold=4GB, max_concurrency=20,
multipart_chunksize=4GB, use_threads=True)
client.upload_file(objstore_filename,objstore_bucketname,mylargefile,ExtraArg
s={'ContentType':'binary/octet-stream'},Config=config)
```

2 Traditional data ingest overview

The input datacube generally consists of spatial data (two dimensions) and frequency data (one dimension), with optionally some polarisation data as well. As mentioned, from a typical ASKAP survey this datacube can be 1 Tb or more in size. Processing of this datacube can be categorised as a "data-reduction" problem.

Most typical AusSRC pipelines make use of the fact that the data-reduction problem is embarrasingly parallel; multiple identical processes can be spawned to compute on different parts of the input datacube, with the outputs being combined for the final results. This is significantly more efficient than having a single process compute over the entire datacube. Hence the single datacube is decimated into *x* subcubes, and *x* processes are spawned to ingest and compute over each of the subcubes.

Consider a source-finding pipeline activity. The compute resources used would comprise of:

1. The data source (in this case an archived datacube within an objectstore at an HPC centre);
2. a compute cluster (eg a SLURM-controlled set of compute instances, either virtual or physical machines);

3. a software process, compiled to run the algorithm(s) on the cluster hardware;
4. a 1 ~ 5 Tb underlying shared filesystem for the cluster.

The shared filesystem is required so that each spawned process (presumably one per compute instance) can read from the same datacube. The generic steps in the procedure would be:

1. Acquire objectstore read credentials;
2. download the datacube from the objectstore to the shared filesystem;
3. start/spawn *x* number of source-finding processes on the cluster;
4. each process reads in a subcube of data from the datacube on the shared filesystem;
5. each process computes across its subcube and writes results to a unique directory on the shared filesystem;
6. a “gather” process collects and merges the processes' outputs to produce the final product(s).

3 Issues with the traditional workflow and using objectstores

The sizes of radio astronomy datacubes coming off current instruments such as ASKAP often approach or exceed 1 Tb - there is no expectation that datacubes from future instruments will be any smaller. The above workflow has an obvious downfall - the (rather large) datacube must be copied over from the objectstore to a shared filesystem prior to starting the pipeline. Additionally this filesystem must then allow efficient concurrent I/O to the same datacube (file) during processing. The pipeline processing quickly becomes an I/O - dominated activity.

Also consider that a large, performant shared filesystem being available to all compute instances within a cluster cannot be assumed - there are significant resource overheads in providing and maintaining such a resource, and many private cloud or supercomputing centres may not provide it. For those that do there is invariably an associated (and significant) cost to the user.

Unfortunately, legacy processes cannot simply read the data from the objectstore. Objectstores do not store data in the same way as more traditional POSIX-style filesystems - there is no notion of “subdirectories “ or “file permissions”, no opening or closing of files, and standard I/O libraries for read/write will not work.

In fact conceptually, objectstores do away with the idea of files altogether; as the name implies, they store objects. An object is stored in a “bucket”, which is a dimensionally “flat” storage concept, as opposed to a POSIX system’s directories and sub-directories. An object is found within the objectstore via a URL, and data is streamed from an object to a destination URL (which *may* be a file on a remote POSIX filesystem).

4 Some benchmarks from using the traditional workflow

Before looking at alternative workflows, we try to quantify the efficiencies of the traditional approach with respect to large astronomical datasets and I/O.

Benchmarks were garnered with the following components and resources:

- an 835Gb spectral datacube (5851 x 5851 x 6668 datapoints);
- the Acacia S3-compliant objectstore (http://projects.pawsey.org.au);
- the Nimbus Openstack cloud-computing resource (http://nimbus.pawsey.org.au);
- a 28-node SLURM cluster provisioned on Nimbus - each node is 32 core, 128Gb of RAM;
- Ubuntu 20 (Linux) operating system on each node;
- a BeegFS 10 Tb shared file system mounted and accessible on each node.

For testing and benchmarking, we used a modified version of the SoFiA-2 source-finding code; this modified version reads the specified subcube of data into memory, reports the elapsed time, and exits. That is, there is no actual processing of the data, and only the initial I/O is run.

The benchmark testing follows the previously outlined workflow:

1. The 850 Gb datacube is downloaded to the shared filesystem;
2. p processes are launched on p nodes in the cluster ($1 < p < 25$);
3. Each process reads a different 33Gb subcube of the datacube (gives total coverage when $p = 25$);
4. Each process logs execution time and terminates.

Each test uses a different value for p to provide the graph below:

From the below graph it is clear that the POSIX shared filesystem allows decent concurrent access up to around 8 processes, above which it begins to become swamped with requests (for 8 processes, a total of 265 Gb is being concurrently read out of the shared filesystem; for 25 process, this extends to 835Gb).

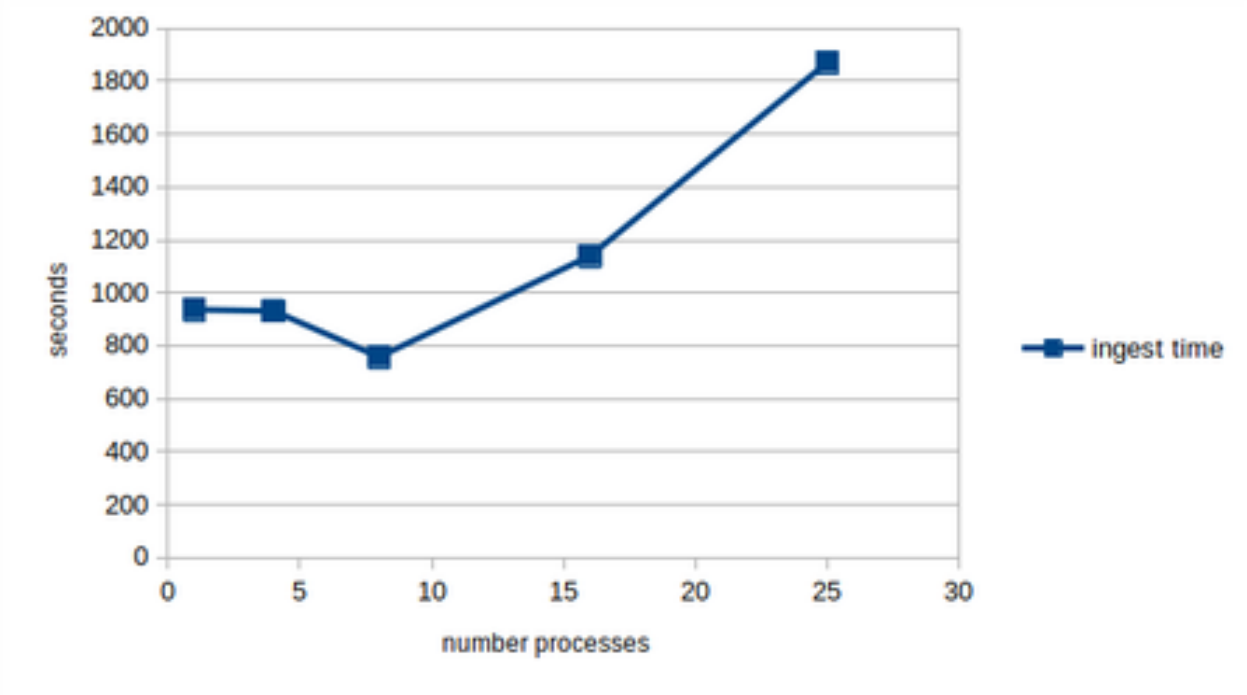

Figure 1: Shared Filesystem ingest time

It is important to note that the above graph does not include the staging time, i.e. the time required to transfer the data from the objectstore to the shared filesystem prior to processing. For this particular datacube, the staging time averaged over several runs, was 9771 seconds, which swamps the read-in times above.

5 Accessing data objects in ObjectStore

There are several ways of programatically accessing an object in an objectstore. The more notable are:

- Using a SWIFT client library: SWIFT is the OpenStack API used for accessing objectstores and is widely used in the public cloud computing environment.
- Using an S3 client library: S3 is the de-facto standard API for interfacing with objectstores as implemented by Amazon.
- Using a pre-signed URL: an objectstore provider can generate a URL with presigned access to an object; anyone with the URL can use standard libraries (eg urllib) to access the object

In this report, we concentrate on the last two methods, as these are supported by Pawsey's Acacia objectstore.

6 Using the S3 interface

The python Boto3 library was used to interface with the Acacia objectstore. Each process directly read the data subcube from the objectstore into memory (33Gb per process, requiring large memory nodes in the cluster), thereby obviating the need to stage any data to the shared filesystem.

The data access performance was monitored for 1, 4, 8, 16 and 25 simultaneous processes, and produced the figure below.

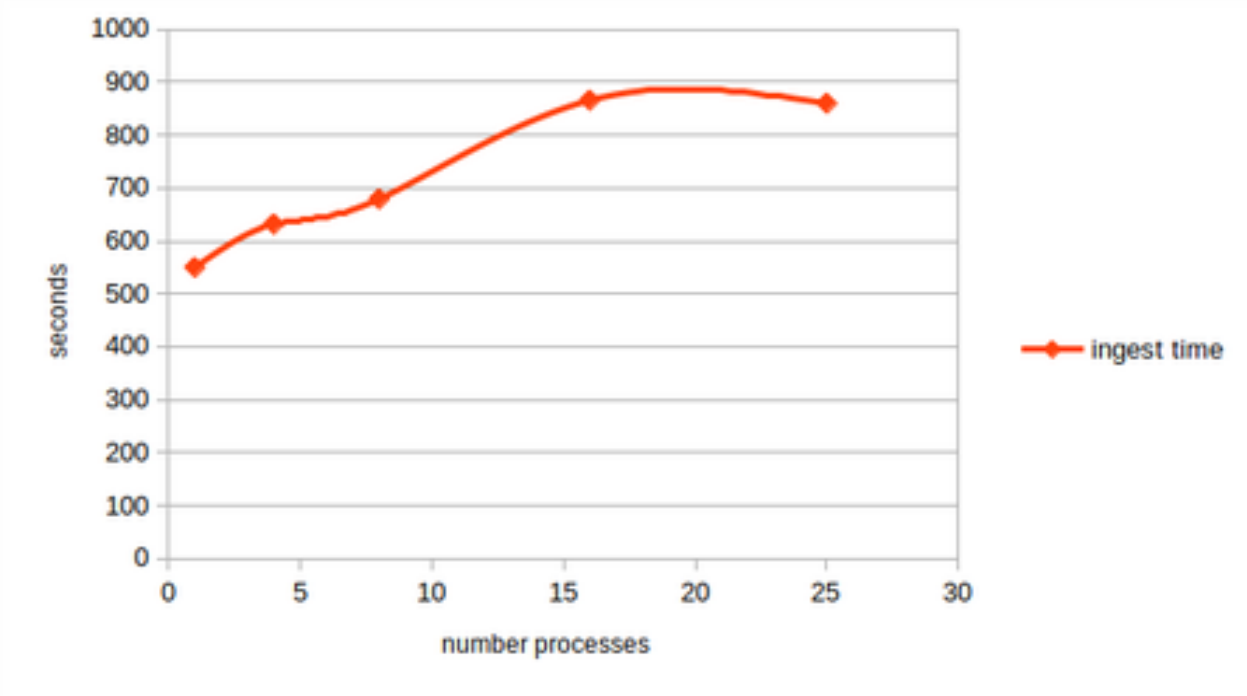

Figure 2: S3 API ingest times

Directly comparing this with the shared filesystem (and including the staging time for the shared filesystem approach) gives the graph in Figure 3:

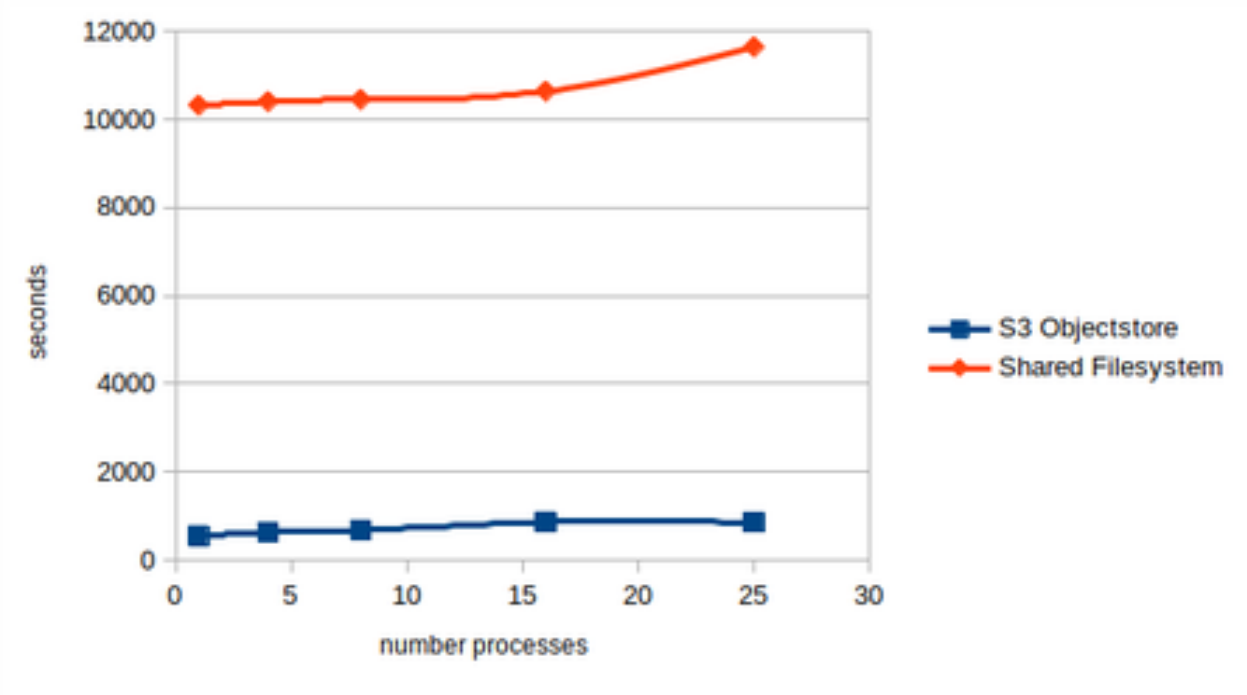

Figure 3: Shared filesystem Vs S3 API read performance

7 Using the Presigned URL Method

7.1 What is a presigned URL?

The typical method for granting users read-access to an objectstore (and the planned method for CASDA), is for the bucket owner to create a “presigned URL” and provide it to the user. A presigned URL (which can be created with the Boto3 library) is a URL for an object within a bucket in the objectstore (some objectstore implementations also allow presigned URL’s for the entire content of a bucket, eg Oracle Cloudstore). A presigned URL is time-limited - that is the creator of the URL puts an expiry date on the URL. After that time, a new URL will be required.

The presigned URL essentially delegates a chosen subset of the owner’s permissions to the user. With the URL, the user can either read or write to the object, without requiring any access or secret keys. All the permissions are encrypted into the URL.

The user can add a header to the URL to, for example, request only a particular byte range of the object. This makes it possible to extract a subcube from a datacube object.

In general, read access using a presigned URL is less efficient than direct access via the S3 API, and performance will be restricted by the bandwidth to the objectstore (around 600Mb/s for the Nimbus/Acacia network). However, streaming to memory can still obviate the need for double handling the data (a.k.a. using a shared filesystem for data staging).

7.2 Testing Python libraries for handling URLs

Python3 provides several libraries for ingesting data via URLs (http); we have tested the following:

- urllib (the oldest of the libraries, and updated to urllib2 in Python2.7)
- requests (the standard library for Python3)
- urllib3 (a third-party library for Python3)

Additionally, there is a restriction to the size of an atomic read operation. Internally, the python libraries use the openssl library to access the URL endpoint; this library has a maximum read size of approximately 2Gb.

We tested all three libraries with read sizes of 1Gb and 1.9Gb, to download 32Gb of contiguous data from the Acacia objectstore. Each read test was done twice - the first downloaded a fresh 32Gb, to avoid the timings being affected by Acacia's caching. The second read was of the same data, so Acacia's caching policies affected the outcome:

Library	Chunk size	Download size	Time (average over 3 runs) - Fresh / Cached
urllib	1Gb	34359738368 bytes	155.7 sec / 89.8 sec
requests	1Gb	34359738368 bytes	153.2 sec / 97.5 sec
urllib3	1Gb	34359738368 bytes	139.1 sec / 86.3 sec
urllib	1.9Gb	34359738368 bytes	154.5 sec / 87.3 sec
requests	1.9Gb	34359738368 bytes	148.2 sec / 92.7 sec
urllib3	1.9Gb	34359738368 bytes	134.1 sec / 61.6 sec

The urllib3 library shows a significant speed advantage, so was used for the rest of the speed tests. ***Note*** that these times cannot be directly compared with the S3 tests above. Those previous tests were for a 32 Gb cube extracted from the larger 853 Gb datacube, ie there is a lot of seeking going on to find the next "row" of the data; here we are simply downloading a contiguous stream of 32 Gb.

7.3 Strategies for downloading subcubes via presigned URL

Fundamentally, retrieving data via the URL is a streaming http GET action. The objectstore does not support random access, so range statements such as:

```
hdr = {"Range":"bytes=%s-%s" % (start,end)}
stream = self.http.request("GET",self.url,headers=hdr)
stream.read()
```

will all follow the same procedure:

1. start at the beginning of the file,
2. read to the “start” position, but dump (ie not return) the read data (see Figure 4),
3. read from requested “start” position to “end” position and return this data in the stream (see Figure 5).

This makes extraction of a subcube from a larger datacube problematic; either a lot of extraneous data needs to be read (Figure 5a), or a lot of small range calls need to be made (Figure 5b).

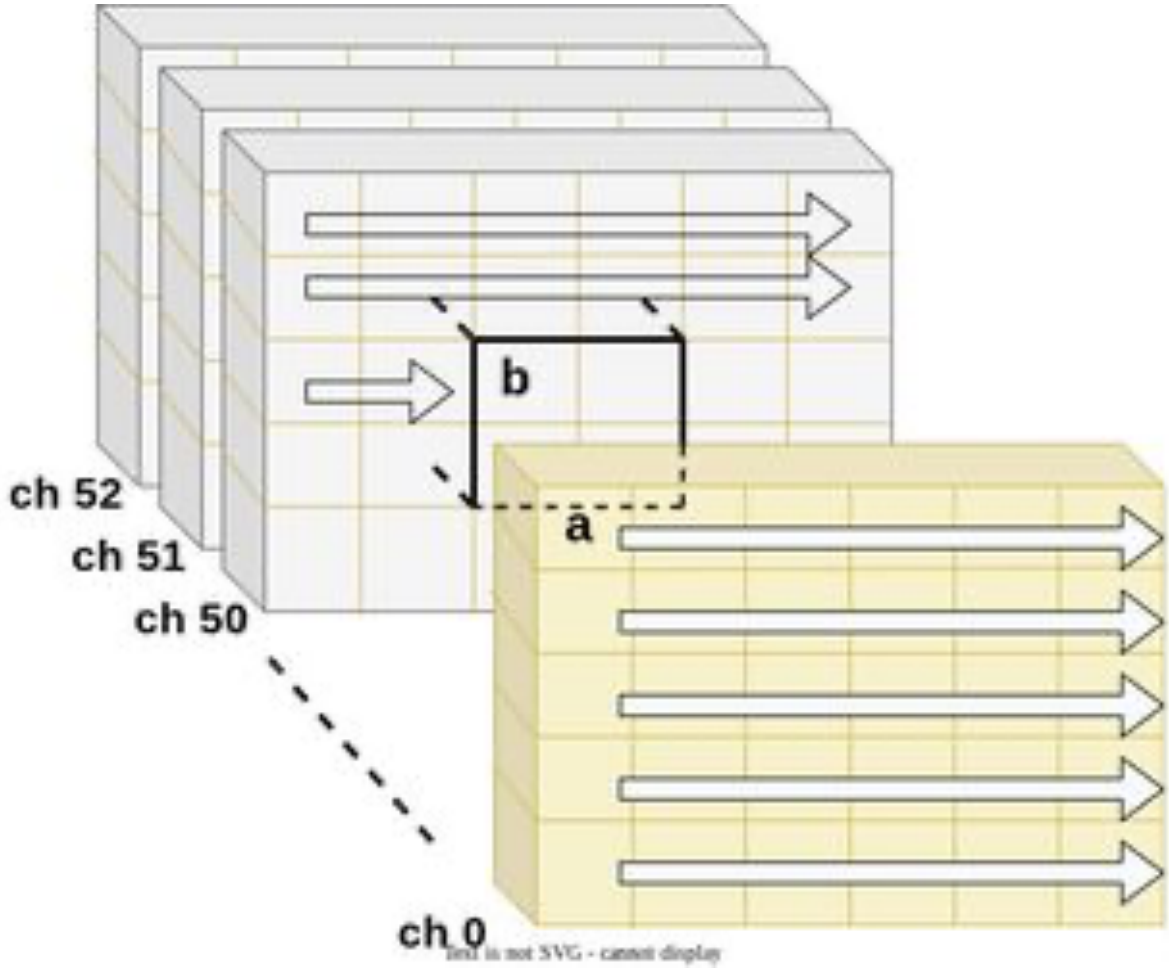

Figure 4: Subcube at (2,3), (2,3), (50,51) - ‘a' is start of datacube. Data is scanned to 'b’ to get to start of subcube; this data is not returned.

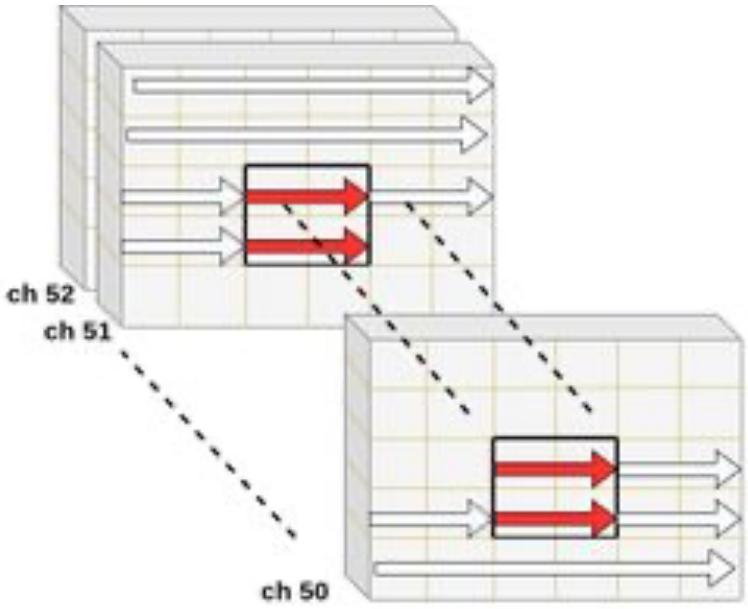

Figure 5a: Reading of the subcube. Group of channel data (red and white arrows) is returned in one read, but only the red arrowed data is required.

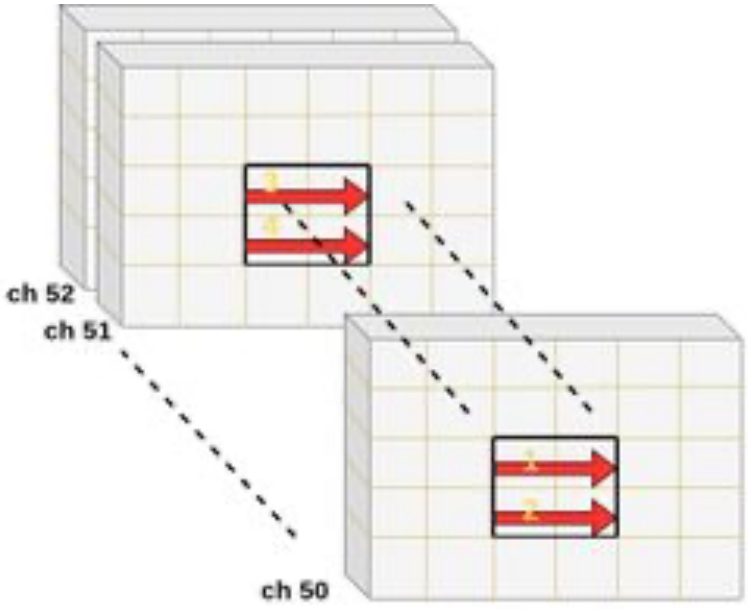

Figure 5b: Alternative reading of the subcube. 4 separate reads are made via 4 separate range calls, but all returned data is required (ie no waste data)

Although at first glance the strategy of Figure 5b may seem more efficient, the small individual reads can mean thousands, or even millions of individual range requests to return the whole subcube, as opposed to a hundred or so for Figure 5a (remembering that there is a limit of under 2Gb per read).

7.4 Testing read strategies for presigned URLs - single instance, single thread

We tested 3 strategies for reading a 34 Gb subcube from the host 853 Gb datacube, using the presigned URL. The subcube consisted of 1893 channels of 2151 x 2151 pixels each (the whole datacube is 6668 channels of 5851 x 5851 pixels):

1. 135 reads of 14 channels at a time (limited by the 2Gb maximum read size), with each read followed by processing and discarding the waste data on the processing node ("Large" read).
2. 1893 reads of 1 channel at a time, with each read followed by processing and discarding the waste data on the processing node ("Intermediate" read)
3. 4 071 843 reads of one row of *x* pixels at a time, no waste data to discard ("Small" read).

The results were as follows:

Strategy	number of reads	size of one read	time (sec)
1. Large read	135	1.79 Gb	959
2. Intermediate	1893	17.6 Mb	3 485
3. Small	4 071 843	8.3 Kb	7 941

From the above table it is obvious that for strategy 3, the overhead of opening a few million range requests far outweighs the benefit of reading ***only*** the required data.

7.5 Testing read strategies for presigned URLs - single instance, multiple threads

The large-read strategy easily lends itself to threading; each 1.79Gb read can be spawned in a separate thread. This is practically limited by the number of threads available, so for instance a thread size of 20 results in around 7 sequences of 20 concurrent reads at a time (for a total of 135 reads). The thread performance for a single instance is shown in Figure 6.

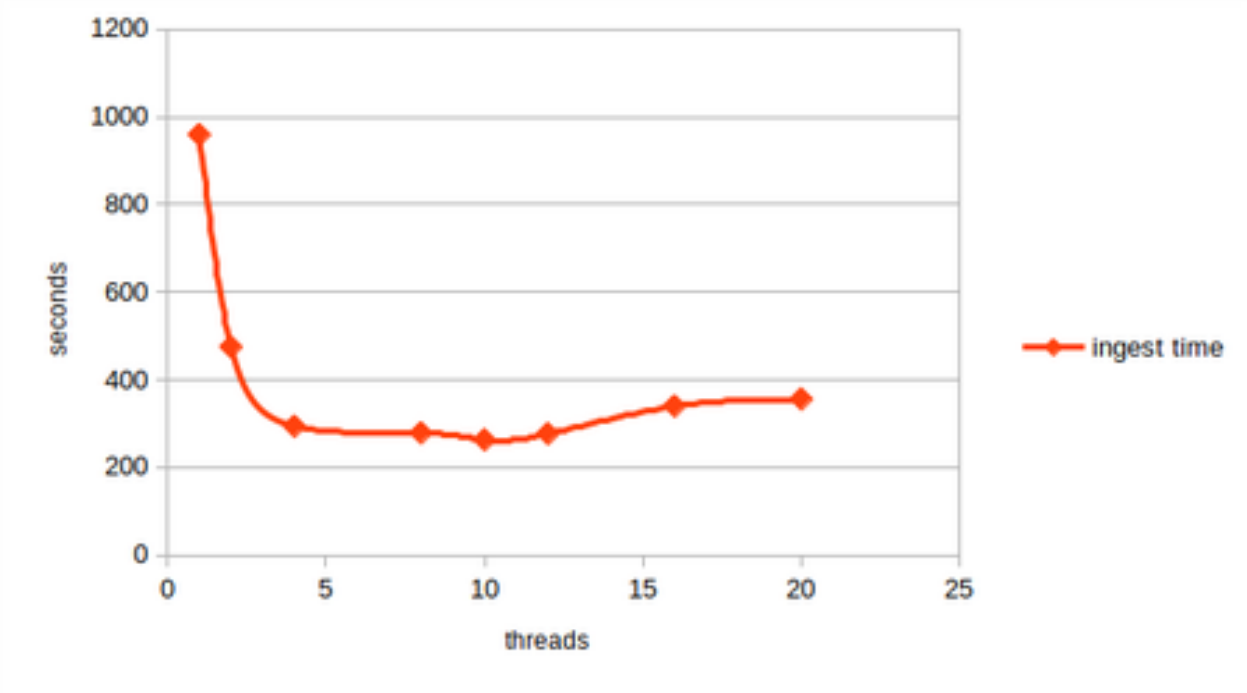

Figure 6: Ingest time versus number of parallel threads for a singe 32Gb subcube download.

7.6 Multiple instances, multiple threads

Using the presigned URL code, the data access performance was monitored for 1, 4, 8, 16 and 25 simultaneous processes of 4 threads each,. Comparing this with the shared filesystem performance produced the figure below.

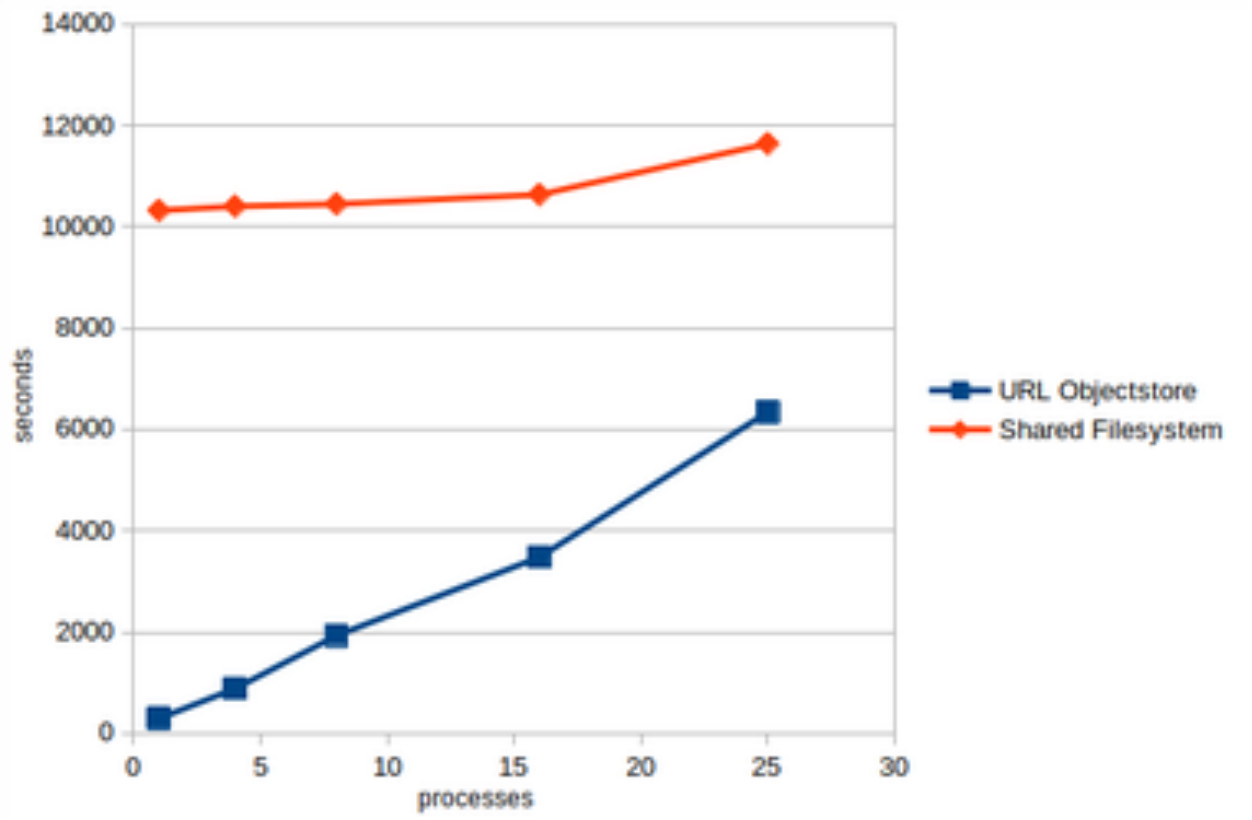

Figure 7: Shared filesystem Vs presigned URL ingest