Considerations and Tradeoffs for Cloud Storage-backed Cutouts from Astronomical Data

Michael St. Clair, Sierra Brown, Chase Million (Million Concepts)
version 1, 2022-09-27

motivation and background

Astronomical data sets are very large (and they keep getting larger), so it is important to find efficient ways to serve data to users. In this document, we discuss the the time, monetary, and resource costs associated with a number of cloud access strategies, and give recommendations for situations in which a particular choice of strategy is likely to be preferable. (**Spoiler**: sometimes transferring less data can take more time due to request overhead).

Specifically, this study investigated optimal methods for taking "cutouts" from astronomical images in the FITS format stored on AWS S3. (We tested with Spitzer, GALEX, HST, Pan-STARRS, and JWST data, along with some synthetic data.) The "cutout" problem is common in astronomy, because individual astronomers are often interested in a few sources of interest within a large image array. Current best-in-class cutout solutions (like the Pan-STARRS cutout service) are generally not backed by cloud storage. When a user requests a cutout, an on-premises server runs some code to slice elements from an array and serves it to the user over the internet. Services of this type are extremely useful but have some limitations. They can accept only a limited number of simultaneous requests before being overwhelmed. They often require institutions to hold data in on-premises storage hardware. Maintaining hardware for services like this across multiple data sets is expensive and tends to make data fusion harder, as it does not allow requests that index multiple data sources at the same time (particularly if these data sets are held by separate institutions).

We believe that backing cutout operations with cloud storage has the potential to provide higher-quality, cheaper, more scalable performance and also lower barriers to interoperability between datasets. However, it also has the potential to provide worse performance at higher cost. This document attempts to help you avoid these pitfalls.

Also note that the cutout problem is not unique to astronomy: many other scientific applications require only a small proportion of the data contained in individual data resources. These data structures aren't unique either: in particular, the fields of terrestrial and planetary remote sensing generate similarly-structured imaging data (n-dimensional binary arrays) and data volumes (with files in the tens to hundreds of megabytes and total data corpuses on the scale of terabytes). This means that most of our results could be extrapolated to other scientific fields and formats,

although implementation might differ (e.g. regions might be defined by geospatial rather than celestial coordinates).

notes on FITS (flexible image transport system)

We assume that our readers are generally familiar with the FITS file format[1], which is ubiquitous in astronomy and allied fields. However, we would like to make a few notes about how the special characteristics of FITS affect its use as a cloud storage format.

FITS is essentially a *container format*: data objects in FITS files are contained within 'extensions' or 'HDUs' (header/data units), and a single FITS file may have many HDUs. Individual HDUs are arranged in distinct, sequential byte ranges within a FITS file. The fact that a FITS file may contain many semantically-distinct data products, laid out in predictable, contiguous ranges, is a major reason that incremental read strategies often work well on FITS files (we discuss this later in more detail).

There are three standard types of FITS extension: images/arrays, binary tables, and ASCII tables. FITS files may also contain a 'primary data array', which is essentially identical in format to an image extension. The FITS format supports storage of tile-compressed images in binary table HDUs[2], which is becoming increasingly popular and is well-suited for subsetting. This investigation was restricted to taking subsets of "images", so we investigated both image/array HDUs and tile-compressed images in binary table HDUs. We did not investigate subsetting in ASCII table HDUs or in binary table HDUs used for other purposes.[3]

Standard FITS image extensions are extremely straightforward data structures: regular binary arrays of up to 999 dimensions (arrays of more than 3 dimensions are rare in practice). They are similar to many other flat-binary array / raster image formats (e.g. ENVI rasters, PDS IMG), although they specifically do not support embedded structures like line prefix/suffix data that tend to make random access more difficult.

---

[1] See IAUFWG 2018, henceforth "FITS Standards", for a complete specification of the FITS format. Unless otherwise noted, all subsequent statements about the permissible features of FITS files implicitly reference this document.

[2] Original specification in White et al 2013.; see also section 10 of the FITS Standards

[3] Prior to v1.0 of the FITS Standards (NOST 1993), FITS also supported a 'random groups' structure that implemented struct/record-like data storage using clusters of regular arrays and pointer-like metadata (Greisen and Harten 1981). This structure did not see wide use outside of radio interferometry and has been deprecated for almost 30 years except for legacy support; its functions have been replaced by binary table extensions. It is outside the scope of this investigation.

cloud storage and subsetting definitions

An *object*[4] is a named sequence of bytes, along with a collection of metadata, contained in and served by an *object storage system* like AWS S3. It is closely analogous to a *file* in a traditional filesystem, and it is usually possible to transfer a file's contents into an object storage system, retrieve the resulting object, and exactly[5] reconstruct the original file (or vice versa). Because of this object-file isomorphism, people often casually refer to objects as files. However, they are not the same. Object storage offers fantastic scalability, but objects provide only a fraction of the features we expect from files, and the features they do provide often perform in qualitatively different ways; moreover, users of object storage systems have no direct access to the filesystem(s) that may (or may not) back an individual object. In this document, we use the term *data resource* to refer to the supercategory of files and objects.

S3 subsetting is a strategy for improving data access efficiency by reading only portions of the total content of an S3 object. Although it is possible to implement S3 subsetting via a variety of interfaces, the underlying mechanism is always the same. AWS S3 supports use of the HTTP Range header field in GetObject API requests. Specifically, a GetObject request may specify, in its Range field, a single contiguous range of bytes; the body of S3's response to that request will contain only that range of bytes from the object, rather than all of the object's bytes.[6] Many web resources other than S3 implement handling for HTTP Range headers. This means that very similar variants of this strategy could be used on most other cloud object stores (such as Google Cloud Storage) or even on traditional web servers. However, these storage backends were outside the scope of our investigation.[7]

A *read operation*[8] is a discrete software action that loads a data resource, or a portion of a data resource, into working memory. Exactly how read operations work varies widely across computing environments and storage interfaces. On S3, read operations are HTTP requests (often abstracted by interface libraries).

In the general case, S3 subsetting is an example of a *random-read strategy*: a method of data access that involves executing multiple read operations to access discontiguous portions of a data

---

[4] This term is dismayingly overloaded in computer and information science, but there is no other general term for the discrete data artifacts contained in and served by object storage systems.
[5] Not counting filesystem-specific metadata like MAC times and permission bits.
[6] See Fielding et al. 2022 for a complete technical specification of the HTTP Range header field, and AWS 2022 for a fuller description of how S3 GetObject API calls implement this specification. Note that S3 does *not* accept multiple non-contiguous ranges in a single GetObject call, and to our knowledge, other major object storage providers don't either. If they did, many of the tradeoffs we describe here might be very different (depending on performance).
[7] If implemented on a cloud object store other than S3, the name *S3 subsetting* would obviously then be inaccurate. We suggest the more general term *cloud object subsetting* for this class of strategies, independent of provider.
[8] This document is about reads, not writes. Analysis of efficient write operations is very different. Also, subsetting is largely inappropriate for write operations to object stores.

resource. In some cases, however, it can act more like a *sequential-read strategy*: a method of data access that involves executing a single read operation to access a contiguous portion of a data resource (but not necessarily the entirety of the resource). Sequential reads are more efficient than random reads in almost every computing context (although the costs of random access differ greatly between contexts), and S3 is no exception. We will discuss cases in which S3 subsetting is "more sequential" later in this document.

tradespace

introduction

Random-read I/O buffering and memory-mapping strategies are widely used on locally-stored FITS files by libraries and applications like CFITSIO, *astropy.io.fits*, and TessCut. The performance improvements offered by these strategies were a motivating factor for this investigation. However, the tradespace for S3 subsetting differs both quantitatively and qualitatively from the tradespace for random reads on locally-stored files. **Strategies that make sense for files do not always make sense for objects.** In this section, we discuss tradeoffs between a variety of costs – primarily operation time, price, and resource pressure – to help readers make informed decisions about cloud access architecture.

This section includes many illustrative examples based on approximate formulae. Please note that these figures are not empirical measurements (unless otherwise stated). To fully assess the quality of a specific access strategy or technology stack, **we strongly encourage collecting metrics in a realistic deployment environment.** As a separate component of this investigation, **we developed a configurable benchmarking suite[9] to facilitate this type of testing.**

transfer speed & request latency

overview

Every read operation takes time. The requester must compose the operation; the signal must reach the storage backend; the storage backend must process it and physically retrieve the data; the response must return to the requester. Depending on the environment, this time cost might be directly described as *access time*, *seek time*, *ping*, indirectly referenced by metrics like *IOPS* (input/output operations per second), etc. We will refer to it as *request latency*. In general, request

[9] The benchmark suite is a submodule of the 'subset' library in the fornax-s3-subsets repository: https://github.com/fornax-navo/fornax-s3-subsets/tree/main/subset/benchmark. Installation instructions are here: https://github.com/MillionConcepts/fornax-documents/blob/main/benchmark_suite_instructions.pdf

latency does not scale with transfer volume. It can be thought of as a kind of transaction cost, or, equivalently, the constant-time component of data read time costs.

All random-read strategies, on any type of storage, trade read operations for transfer volume. You must perform extra read operations to access some portions of a data resource while skipping others. **If these extra read operations are more expensive than it would be to read the whole data resource into memory, then they are not a good idea.** You should just read the whole resource. Attempts to optimize read strategies often center on finding this crossover point.

This tradeoff is sharp and unforgiving in the cloud, **because read operations on cloud objects are very slow relative to locally-stored files** (or files in many cloud filesystems, including AWS EBS[10, 11]). Data access time for locally-attached solid-state drives (SSDs) is measured in microseconds; mean SSD request latency is typically in the 70-250 us range (and can be significantly better with high-performance hardware in optimized scenarios). Fast magnetic hard disks (HDDs[12]) have mean request latency around 1-5 ms; few modern HDDs have mean request latency > 15 ms (and HDDs even that slow would likely be in "cold" backup roles). Conversely, the *very-best-case* request latency for S3 is ~19 ms (a HeadObject or ListBucket request for a single frequently-accessed object, issued from an EC2 instance to an S3 bucket in the same AWS Region, *not* counting the caller's handling time for the request or response). 30-80 ms request latencies are more typical even in quite good cases, and 200+ ms request latencies are very common for calls made from outside of the AWS ecosystem, even for callers in the same general geographic region with good internet connectivity.

To summarize: **read operations on S3 objects are typically 1-3 orders of magnitude slower than on local files. This makes request latency, in most cases, the primary driver of differences between local and cloud read-strategy tradespaces.**

Transfer speed is data volume transferred per unit time. In local storage contexts, maximum transfer speed is often called *throughput*; in networking contexts, *bandwidth*. Confusingly, these two terms are often used to refer both to the maximum speed of individual transfer operations, and to the total speed a particular channel or interface can support across all simultaneous transfers. This distinction is extremely important in networked contexts that use distributed delivery networks (like S3), so in this document, we use 'bandwidth' and 'throughput' to refer only to maximum total speed.

---

[10] Predicting EBS latencies is difficult because of EBS's deep, opaque virtualization and networking stack. However, the figures we give here for SSDs and HDDs tend to be in the correct ballpark for SSD-backed EBS volume types (e.g. gp2 & gp3) and HDD-backed EBS volume types (e.g. sc1 and st1) respectively, with the notable exception that small (<1 TB) HDD-backed EBS volumes can have *extremely* poor request latency, up to 100ms in some cases.

[11] Not *all* cloud filesystems offer fast read operations relative to S3. For instance, many AWS EFS configurations have fairly poor request latency.

[12] It is more difficult to predict HDD than SSD request latency due to the complex interactions between actuator arm and rotational latencies. It can vary a great deal between differently structured and sequenced operations and filesystems on the same physical drive.

We emphasize that request latency is the primary difference between local and cloud read strategies because most information technology professionals – including us – initially assume that transfer speed will be the bottleneck in any use of cloud object storage, and that whatever it takes to reduce total transfer volume is probably worth it. This intuition is driven by the fact that it is very common for us to wait impatiently for large files to finish uploading or downloading – files we could copy in mere moments between local filesystems – but *uncommon* for us to attempt to upload or download thousands of small files at once. Transfer speed and bandwidth *are* important. But latency is almost always a *bigger difference* between data resources stored locally and data resources stored in the cloud.

Moreover, as read operation volume increases, the marginal data volume savings per operation decreases. One contributor to this is the fact that each read operation has volume overhead – metadata contained in the response – and AWS APIs tend to be fairly verbose. A request for even a single byte of an object returns about a kilobyte of metadata. In other words, **single-minded pursuit of data transfer savings is usually self-defeating**. For instance, taking many cutouts from a single image is usually a bad idea; more generally, **subsetting is usually not appropriate for use cases that require a large proportion of the area of each accessed image** (how large a proportion depends on the image and environment).

Also note that **request latency and transfer speed are not the only contributors to the time costs of additional read operations.** For instance: it is often more computationally expensive to handle many small chunks of data than a single large chunk of data, each chunk of data must be transferred between various local memory resources, composing API requests and parsing API responses takes time, and so on. Transfer optimizations do not necessarily improve this situation; in fact, any transfer optimizations in place may themselves incur time costs. Although the subsetting time budget tends to be dominated by transfer time costs, these factors are also important.

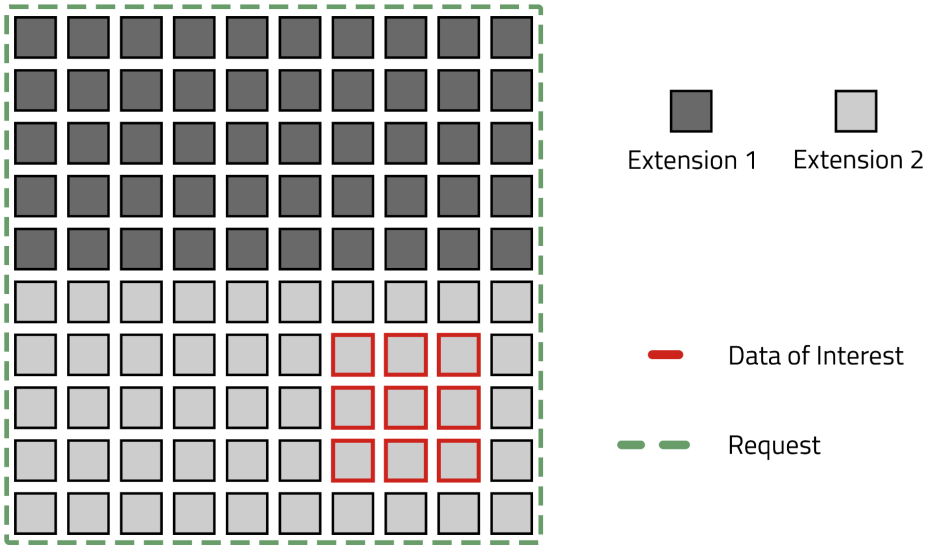
In order to decrease the *time cost* of request latency, parallelization can be implemented. S3 handles parallel requests well, but not infinitely well. The number of requests per second it will accept is variable, partly dependent on internet weather, affected by complex and secret serverside load-balancing, and overall extremely difficult to predict. **If aggressive retrieval parallelization is in use, these other contributors can even grow to dominate the time budget**. Parallelization factors are very complex; we discuss them in more detail in the 'parallelization' section below.

simplicity/atomicity <-> complexity/subsetting continuum

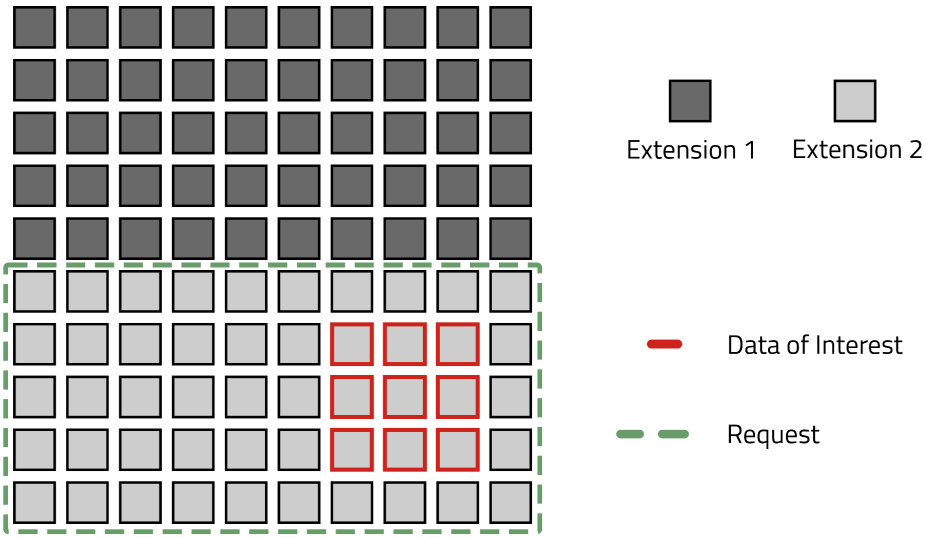
There are four basic types of data retrieval options for FITS arrays stored in S3 objects. We list these in order from the most simple and atomic (transferring entire objects without slicing) to the most complex and intensive (subsetting only the specific bytes of interest). We describe

uncompressed arrays here, but our comments largely also apply to tile-compressed arrays, on the array-tile rather than array-element level (see “tile compression” below for more discussion of tile compression).

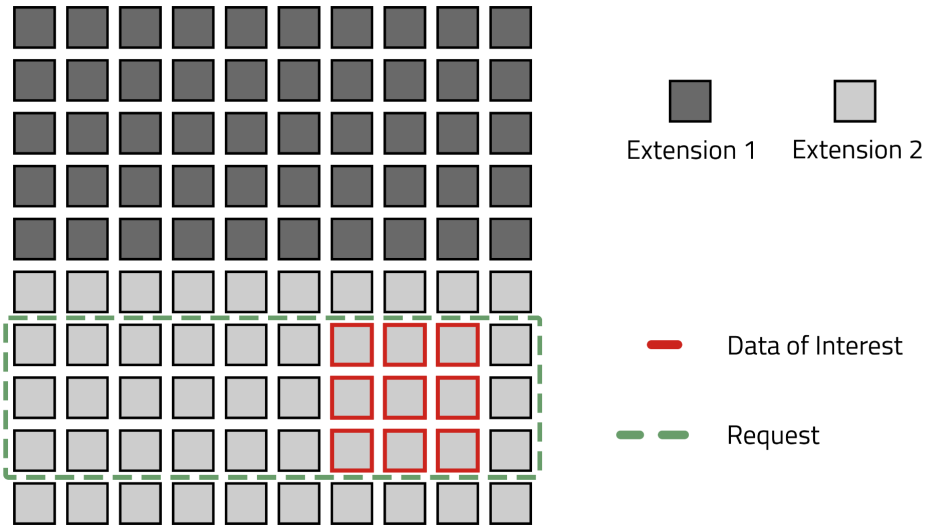
1. *Retrieve the entire S3 object*: This method only takes **one API call** (a GetObject request) but **transfers the highest volume of data**.



2. *Retrieve only the extension of interest*: This method will usually take at least **4 API calls**: a HeadObject request for the object’s S3 metadata, a GetObject request to read the SIMPLE card and primary HDU, and a GetObject request to read the extension’s header and figure out how big it is, and then a GetObject request to retrieve the extension. (If we are interested in an extension after the first one, more requests, one per extension before the one of interest, will be required to read each header and ‘seek’ into the object, unless the request is meticulously constructed in a non-generalizable way. Conversely, if the array we want is actually in the primary HDU, it may require only 3 API calls.) **This method transfers less data than retrieving the full object** (assuming there are multiple extensions).



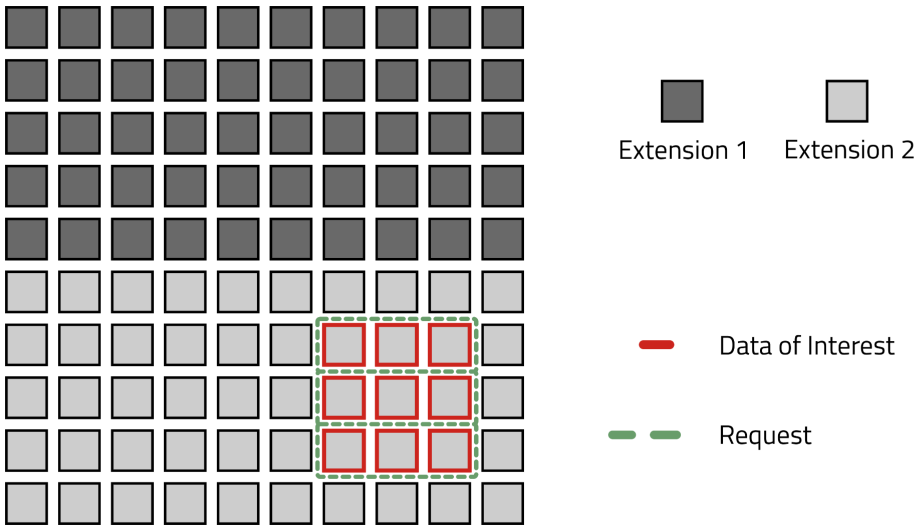
3. *Full-line subsetting*: This method takes advantage of the row-wise layout of FITS arrays and pulls every line that contains any array elements of interest. Because FITS files are in row-sequential formats, this can be accomplished with a single retrieval request that pulls all the rows of interest at once. This would not be possible for a request that pulled a full column of data; it would have the same result as pulling the whole array. The situation becomes more complex with >2D arrays[13]. This usually requires the same number of requests as retrieving the full extension; the final GetObject request retrieves just the lines of interest rather than the whole extension. **This method transfers less data than retrieving the full extension**.



4. *Partial-line subsetting*: This method only pulls the specific bytes of interest, but **requires an additional GetObject request for every line,** as the byte ranges are noncontiguous between lines. This is in addition to the (at least) 3 preliminary requests: HeadObject for

[13] Things can get really bad really fast if you take a cutout from a >2D array in a "wrong" direction.

metadata, GetObject for primary HDU, GetObject for the extension's header. **While this method transfers the least amount of data, it has the highest request overhead.**



quantitative approximation

A simple, but useful, approximation of transfer time can be given by the formula:
transfer time = request count * request latency + transfer volume / transfer speed.[14]

Consider a FITS file with a minimal primary HDU and 3 image extensions, each of which is a 1500x1500 2D array in 32-bit floating point (much like a smaller Hubble drizzle image). You want a 40x40 cutout from the first image extension. The FITS file is stored as an S3 object, and you are in an environment with 30-ms GetObject and HeadObject request latency[15] in which S3 will offer 80 MB/s transfer speeds on individual streams. (These are fairly typical figures for S3 to general-purpose non-burstable EC2 instances, e.g., the m6i family, within an AWS Region. In practice, these are not constant values and may even change over the course of an operation.)[16]

---

[14] This model ignores many, many factors, such as volume-dependent variations in speed and latency. However, it is often a decent first-order approximation of these factors of the tradespace.
[15] Adding a range field to a GetObject request does not, in general, increase latency.
[16] Specifically, this is a realistic transfer speed for short / small transfers. Large within-Region single-stream transfers from S3 to general-purpose non-burstable EC2 instances tend to peak at values closer to 200 MB/s. However, individual streams typically take several seconds to "spin up" to their full speed, so 200 MB/s is unrealistic for transfers with sub-second durations. Similarly, this request latency is realistic for objects that are accessed very frequently. If the objects are *not* accessed quite frequently, these figures are overly optimistic for subsetting, although request latency might drop to this value over the course of a single intensive subsetting operation as S3 moves the object into a higher caching tier. S3's variable request latencies, along with the fact that S3 transfer speeds are partly a function of total volume transferred per stream, are just two of many factors that make it difficult to compose truly accurate predictive models of S3 subsetting.

Retrieval options:

1. **Retrieve the entire S3 object (don't subset).** The object is ~27 MB (~9 MB per array; primary HDU & headers have trivial volume). If the object's name and general properties are already known, retrieving it will take only one API call. Total transfer time is 30 [ms / request] * 1 [request] + 27 [MB] * 1/80 [s/MB] * 1e3 [ms/s] = **~370 ms**.
2. **Retrieve only the extension of interest (good in this case).** The first extension is ~9 MB in total. As described above, this will take 4 API calls.
   Transfer time for each array element is 4 [B / element] * 1 / 80 [s / MB] * 1 / 1e6 [MB / B] * 1e3 [ms / s]: **5e-5 ms/element.** Then total retrieval time is 30 [ms / request] * 4 [requests] + 1500^2 [elements] * 5e-5 [ms/element] = **~230 ms. Retrieving single HDUs from a FITS object is very often effective.**[17]
3. **Full-line subsetting (better in this case).** Grabbing every 'line' / row that contains any portion of the cutout again requires 4 API calls. Total time is 30 [ms / request] * 4 [requests] + 1500 * 40 [elements] * 5e-5 [ms/element] = **~120 ms**. This is a meaningful savings over transferring the whole extension of interest. Whether or not saving 110 ms is important depends on the application, but **subsetting is almost always effective for use cases that only require single small cutouts.**
4. **Partial-line subsetting (bad in this case).** Retrieving just the bytes of the cutout will take 30 GetObject requests in addition to the 3 preliminary requests, one for each line / row: 30 [ms / request] * 33 [requests] + 40^2 [elements] * 5e-5 [ms/element] = **~990 ms.** Partial-line subsetting hurts a lot in this case.

further time cost considerations

object/array size

Subsetting becomes more effective as object (and array) size increases. If each of the image extensions were 4200x4200 64-bit arrays, each extension would be ~128 MB, the entire object size would be ~384 MB, and transfer time per element would be 1e-4 ms. Transferring the entire object would take ~4830 ms[18]; atomically transferring the first extension would take ~1880 ms; full-line subsetting for a 40x40 cutout would take ~140 ms; partial-line subsetting would take ~990 ms. The pessimistic crossover point between atomic extension transfer and full-line subsetting would occur between 38 and 39 cutouts. This is because transfer time for each line is almost 6 times longer than in the smaller array, so saving total lines transferred remains time-efficient up to the point you are transferring about a third of the array – and this is *without* any readahead optimizations, and assuming an extremely bad case for cutout distribution. But it is still the case that **if you are using a very large proportion of an array, subsetting on a more granular level than retrieving an entire HDU is generally unhelpful.**

[17] Subsetting at the extension level basically treats a FITS file as a *dataset* or *virtual filesystem*. This category of architecture is very popular in managed cloud processing frameworks (Spark, etc.)
[18] This is a bit pessimistic, as speed will often increase a little by the end of these larger transfers.

This observation is really just a corollary of the principle that request latency does not scale with data volume. While the small files have the same overhead request latency, the larger a file the larger a portion of the overall time budget transfer speed takes up. However, **most real-world FITS files aren't this big, and the ones that are tend to be stored in compressed formats.**

number of cutouts

The basic character of these read options doesn't change if a use case requires multiple cutouts per object, but performance tradeoffs can change, especially if requested data ranges may contain data from more than one cutout (as would be the case in option 2 if multiple cutouts were needed from the same extension, or option 3 if multiple cutouts shared a line). The higher the percentage of the total array is covered by cutouts, the more likely it is that a simpler read technique will be more effective. Basically, **subsetting gets less effective if you want more than one cutout.** The 'crossover' point in the small-array quantitative example between 'full-line' subsetting and retrieving the full HDU (assuming no cutouts share the same lines) occurs between 3 and 4 cutouts: 4 cutouts will take 30 [ms / request] * 7 [requests] + 1500 * 160 [elements] * 5e-5 [ms/element] = **~220 ms.**

Transfer-time optimizations can be made at intermediate layers of the stack. If the cutouts share any lines, the use of in-memory caching or automated request chunking can reduce the total number of bytes transferred and/or reduce the number of required requests. Fixed or adaptive readahead strategies can also reduce request volume (sometimes not just for cutouts). **In the best empirical cases, these optimizations allow request volume to grow roughly as the square root of the cutout count rather than linearly.**

No matter how many clever optimizations are in place, as the number of required cutouts and/or proportion of array volume covered by cutouts go up, the less effective subsetting is vs. transferring the whole array. Similarly, there is **no one-size-fits-all time-transfer optimization; a specific optimization won't always be helpful and may even hurt**. For instance: consider a retrieval backend set to an 8 MB minimum readahead size, but with no memory caching (or ineffective memory caching, or simply bad luck[19]). This would be *excellent* in many cases. However, even on that big 4200x4200 64-bit array, 16 cutouts would transfer *at minimum* the full volume of the extension in addition to their request latency overhead. More would always be worse than transferring the full extension in *both* volume and time. On the 1500x1500 32-bit array, this crossover point would occur at only two cutouts! **Selecting – and implementing – transfer-time optimization strategies is a nontrivial problem.**

scientific use case

The time savings offered by subsetting must be assessed in the context of the overall time budget of a particular scientific use case.

[19] Even with perfect memory caching, with strict 8 MB readahead, you must always retrieve at least 8 MB per request. Thus, even if you are missing a single byte from the next cutout, you'll still have to retrieve 8 MB to add that single byte to your cache.

Good use case for subsetting: Assume our 4200x4200 64-bit array is a single tile from a full-sky survey that includes 250000 total images. Based on analysis of another dataset, an astronomer has identified 5000 sources of interest, scattered across the entire sky such that each tile of our full-sky survey contains at most one source. The astronomer would like to perform simple aperture photometry on each of these sources to add spectral depth to their existing analysis; these photometric operations will each take 50 ms. Retrieving all 5000 S3 objects and performing aperture photometry on a single source from each would take almost 7 hours. If this analysis is blocking (in the project management sense), this might be an entire day of work lost to staring listlessly at a progress bar. Conversely, retrieving just a cutout around each source of interest will only take ~15 minutes; the astronomer can just take a coffee break and return to *doing science*.[20]

Bad use case for subsetting: An astronomer would like to perform complex Bayesian photometric analysis of a field that covers just the "upper" third of our 4200x4200 64-bit array. This analysis will take roughly an hour to execute. Retrieving the upper third of that array would take ~708 ms, a savings of over 4 seconds compared to just retrieving the whole object – *amazing!* However, this retrieval time savings reduces the overall time cost of the analysis by barely over a tenth of a percent. Subsetting is likely not worth the bother.

bandwidth limitations

Intuitively, subsetting should be more time-effective in low-bandwidth environments. **However, request latency often remains equally important in bandwidth-constrained environments, because factors that decrease transfer speed from S3 also tend to increase request latency**. For instance, from the computer I am working on right now in my office, I get ~20 MB/s single-stream transfer speeds from S3 buckets in us-east-1, and best-case request latency ~110 ms. Request latency and transfer speed are both roughly 4 times slower than in the EC2 examples above. Although the situation is worse overall, the *relative* contributions of these factors to my overall time budgets are about the same as they are on EC2.[21]

request and transfer volume: monetary and resource costs

reasons to care i: money

[20] This is somewhat optimistic, because 30ms response time assumes that all of those files are regularly-accessed and in one of S3's higher (and completely opaque) caching tiers. A more realistic assumption of 80ms response times takes us up to a little over 30 minutes. (Perhaps the astronomer can *walk* to a nearby coffee shop.)

[21] As above, this analysis is true for small transfers, 'small' meaning < ~100MB – a cap well above the transfer sizes involved in most subsetting operations. Transfer speed differences between my local environment and EC2 would be larger for larger arrays – especially full extension/object transfers – and it would come to overshadow request latency in my overall time budget.

There are many reasons organizations choose to implement portions of their IT infrastructure in vendor cloud services rather than on-prem equipment, but monetary costs are often a major factor. Switching from on-prem to cloud services tends to replace fixed costs with variable costs. These variable costs are often smaller, but are highly multifactorial and can be difficult to predict.

An overview of all the cost considerations involved in S3-backed astronomical data services is beyond the scope of this document. However, it is useful for an organization to consider how FITS subsetting strategies might affect the bottom line. Thus, we provide a detailed analysis of how it might modify individual drivers of cloud costs.

In most applications, the largest drivers of S3 costs are *stored data volume*, *storage tier*, *API request volume*, and *transferred data volume* (often called *data egress*). Because data access strategies like subsetting have no first-order effects on stored data volume or tier, we assume in this document that stored data volume and storage tier are non-negotiable quantities: the organization holds the data it holds, and it must serve them with high availability (meaning in the S3 Standard Tier)[22]. **This leaves only request volume and data egress as points of discussion.**

cost of data egress

It usually costs money to transfer data out of AWS (including S3). As of 2022-09-15, assuming that a hosting account transfers over 150 TB of data from S3 to the open internet each month, the marginal cost of transferring 1 MB to the open internet is 5e-5 USD. (S3 egress fees scale down from 9e-5 to 5e-5 USD/MB at three separate volume/month cutoff points.)

However, not all AWS data transfers directly cost money. Data transfer within an AWS Region is free, and data transfer *between* AWS regions is cheaper than transfer from AWS to the rest of the internet. **If you can keep all your work on AWS and in a single Region, you can avoid egress fees, although bandwidth is still a limitation, which may indirectly drive costs**. Additionally, many scientific archives receive discounts or waivers for S3 egress (and storage) fees, either through membership in the AWS Open Data Sponsorship Program or individually-negotiated sponsorship agreements. Readers from these institutions may find this discussion less relevant to their concerns. However, the terms of these agreements generally include a variety of caps and carveouts – particularly for non-public data holdings – so they do not reduce egress costs to 0 even in the best case. We also feel it would be unwise to assume that the Open Data Sponsorship Program will be in existence forever. **So, even if you're in an AWS sponsor program, you should probably keep an eye on *potential* monetary costs of egress.**

In cases where data egress fees are absent or waived, data transfer price depends only on request price. As we saw earlier, **more complex subsetting requires more API requests, so**

[22] Storing objects in other tiers can enormously change these tradeoffs, both quantitatively and qualitatively. One major difference is that data *retrieval*, not just egress, may have associated charges. Requests may also be much more expensive.

without data egress fees, subsetting will *always* increase total transfer costs. This means that if egress fees are not a consideration, you should strike *any* monetary transfer savings from the examples below. However, you might still save time, and – as we also discuss below – time isn't always monetarily free.

cost of requests

Although egress is free in many cases, requests are almost never free. Requests made from an EC2 instance to an S3 bucket in the same Region cost the exact same amount as requests made to that bucket from the open internet. Sponsorship agreements rarely waive request fees. Request costs can end up being *trivial*, but they are almost never *absent*. **Not all S3 requests cost the same amount of money.** S3 request charges come in two 'tiers' depending on the specific API call. **In general, write-type requests cost an order of magnitude more than read-type requests (with the notable exception of LIST requests).** For instance, as of 2022-09-15, for objects stored in S3 Standard Tier in US-East-1, each GET and HEAD request costs 4e-7 USD, while each PUT request costs 5e-6 USD.

A simplified model of transfer (egress + request) price for a particular read operation is given by the equation below:

transfer price = number of requests * price / request + transfer volume * price/unit volume

tradespace examples/synthesis

There are three main regions of the time-volume-complexity tradespace. When required cuts on the data are simple, subsetting techniques save both data egress (volume) and time. However, there is a crossover point after which increasing subsetting complexity continues to save volume but begins to cost time[23]; finally, there is a 'ceiling' past which more complex subsetting only costs time (and in practice usually also volume). For an extreme case, subsetting each element from an array can never save volume over transferring the whole array; this is the theoretical lower bound for the density/complexity 'ceiling'. However, various sources of overhead (such as the data volume of the response headers and the higher number of API requests required) will always make subsetting cost more in *both* time and egress than just transferring the whole array *well* before this theoretical lower bound.

example 1: small FITS objects

Consider retrieval options (on the small array) using our simple model above, let's assume our user wants to retrieve a single 40x40 cutout from each of 5000 such objects. **Because S3 subsetting (like all random-access strategies) trades transfer volume for request volume, it also trades egress fees for API request fees.** The table below describes the volume/time/price tradeoffs for

[23] As discussed above, this is because request latency does not scale with transfer volume.

this example. Subsetting definitely saves money in this case; however, **partial-line subsetting is slower and also not cheaper than full-line subsetting.**

Retrieval option	**Transfer volume *(per object)***	**Time *(per object)***	**Cost *(total 5000)***
Full object	27 MB	370 ms	~$6.50
Single-HDU	9 MB	230 ms	~$2.00
Full-line subsetting	0.24 MB	120 ms	~$0.07
Partial-line subsetting	0.0064	990 ms	~$0.07

example 2: multiple cutouts

Again on 5000 of these FITS objects, consider varying required numbers of cutouts per array, weighing full-extension vs. full-line retrieval. The table below describes the volume/time/price tradeoffs between single-HDU and full-line subsetting for 1, 2, 5, and 20 cutouts (taking additional cutouts does not change volume, time, or price for single-HDU subsetting). We would hit the request-volume price crossover point between 33 and 34 cutouts / object. As with earlier high-complexity examples, this is illustrative but not realistic: in practice, without a laboriously-designed special-purpose codebase, taking that many cutouts from such small arrays would make *everything* worse.

Retrieval option	**Δvolume *(per object)***	**Δ time *(per object)***	**Cost *(total 5000)***
Single-HDU	0 MB	0 ms	~$2.00
1 cutout (full-line)	8.76 MB	-110 ms	~$0.07
2 cutouts (full-line)	8.52 MB	-70 ms	~$0.13
5 cutouts (full-line)	7.80 MB	+30 ms	~$0.31
20 cutouts (full-line)	4.20 MB	+472 ms	~$1.25

time as a monetary cost

In some cases, time is also a direct driver of monetary costs. Two common cost-of-time categories are work hours and instance hours. Assume our hypothetical user is “on the clock”, billable at $80/hr. In the 20-cutout case in Example 2, we saved a total of ~75 cents at a time cost of ~40 minutes. If our user is *very* lucky, this might cost them only about 5 billable work-minutes of wasted time in context switching (checking progress bars and so on). In this near-best-case scenario, we’ve ended up *costing* about $6.70 to save $0.75. Even if that’s our $0.75 and someone else’s $6.70, repeated experiences of this kind may make them disinclined to use our services. **We**

saved < 40% on this AWS transaction, but imposed a > 500% brokerage fee on the research community (even before considering secondary effects of lost productivity).

As for instance hours, if you execute subsetting operations on an on-demand EC2 instance that save S3 costs but increase running time, those S3 cost savings are offset by the instance's price per hour. For instance, if this operation is running on an m6i.2xlarge instance (admittedly overpowered for this application) with an on-demand rate of $0.38 cents an hour, we have incurred additional total AWS fees by choosing full-line subsetting. (This is entirely separate from any marginal compute charges we might incur with some instance families.)

reasons to care ii: resource constraints

bandwidth and rate limitations

Every server[24] has bandwidth and request rate limitations. If you are doing more than one thing at a time on a server – or if multiple users are running tasks on the server – the server can run out of bandwidth or have S3 throttle its request volume even if individually each are well below these overall resource constraints. Workload prediction and optimization can enormously affect the performance tradeoffs involved in subsetting, and overall workload can also be an indirect driver of monetary costs[25]. For instance, even if you are transferring data only from EC2 instances to S3 buckets in the same region – and so incurring no direct egress fees – multiple high-transfer-rate tasks may nevertheless saturate the instances' network interfaces and degrade performance to unacceptable levels. This might force you to increase your number of running instances, lengthen the duty cycle of your on-demand instances, or purchase more expensive instances with larger amounts of available bandwidth – even if your existing instances have more-than-adequate system resources of all other kinds. Apparently-slow subsetting operations might be worth it to take some of this load off.

memory

Most FITS images are not large enough that simply loading them into memory from local storage presents serious memory pressure considerations on modern personal computers. There are certainly exceptions, but memory mapping and similar I/O buffering strategies can usually handle those exceptions. Complex processing pipelines that involve in-memory copies can of course put pressure on working memory, but optimized retrieval strategies do not improve those situations: if you only need part of the array, you can just discard the rest from memory before executing your pipeline. **So, most of the time, S3 subsetting isn't any more efficient in terms of working memory requirements than just retrieving an entire object, scratching it to local storage, and loading the portions of interest into memory.**

[24] Or other computing platform, including 'serverless' agents like AWS Lambda functions.
[25] A full discussion of workload prediction and optimization is beyond the scope of this document.

However, the scratch-to-disk strategy comes with downsides. **Local storage volumes have their own read/write overhead, so the operation will be overall slower.** You also have to implement the scratch operation, which adds code or workflow complexity. And, of course, the space has to be available, which can rule out some desirable architectures. For instance, many astronomical analysis pipelines that operate on small sections of arrays are not computationally intensive, so it can be appealing to execute them on extremely cheap cloud servers. However, if those small array sections come from 120 GB TESSCut cubes and you're planning to scratch those cubes to disk, the cost of maintaining local storage volumes for your servers will quickly come to overshadow the cost of the servers themselves. This gets worse if you'd like to use a serverless architecture based on AWS Lambda: scratching even one of these files from S3 to disk would simply be *impossible*, because Lambda local storage maxes out at 10 GB. If you retrieve array sections with subsetting techniques, however, these problems go away. This is an extreme example, but it illustrates the fact that, **in many situations, using subsetting *qualitatively*, not just quantitatively, improves the capabilities of compute resources, and, by extension, makes a greater variety of IT architectures feasible.**

parallelization

Analysis of performance considerations involved in parallelized retrieval operations is difficult and involved. However, because S3 request parallelization is effective and ubiquitous, this document would be incomplete without a brief discussion of how parallelization can influence the tradespace.

vocabulary

In this section, in order to keep our discussion relatively general and avoid overloading terms, we use the following vocabulary:

- *Workers* are simultaneous or pseudo-simultaneous software execution contexts; this term is an implementation-agnostic catchall for terms like 'process', 'thread', 'coroutine', 'worker', etc. used in specific languages and computing environments.
- *Threads* are simultaneous hardware execution contexts, whether provided by multiple physical processor cores, simultaneous multithreading technologies, or whatever else.

Our discussion to this point assumed that all S3 requests and retrievals occur in series. It is perfectly reasonable to execute S3 operations in series, and this has some advantages in code complexity and stability. However, because of S3's content delivery infrastructure, it deals very well with parallel requests, to the point that **Amazon explicitly recommends executing S3 operations in parallel as a first-line optimization strategy.** Unless you are in a very bandwidth-starved environment, you can typically achieve a higher total transfer rate across

multiple transfers; similarly, there is a large domain where request latency scales very little with request volume[26].

S3 calls parallelize so well that parallelization is often used for full-object retrieval. Many S3 interfaces use adaptive multipart upload/download strategies to parallelize transfers, looking for the same sorts of crossovers between request latency and transfer time we considered in our discussion of serial subset retrieval. This also means that **parallelization is built into many off-the-shelf general-purpose cloud access technologies,** so parallelization does not necessarily incur serious code complexity / fragility or development time costs (although it can).

Intense parallelism tends to be most effective when there are lots of operations to perform. The overhead of managing lots of workers becomes relatively more expensive when there isn't much for each worker to do.

limits to parallelism

When executed in environments with high connectivity to S3, parallelization of subsetting operations is often limited by system resources, especially CPU resources.[27] This fact is counterintuitive for many IT professionals, who tend to assume that networking resources will be the bottleneck for networked operations. But subsetting can use networking resources very efficiently, and performing read operations takes compute resources. At minimum, the requester must compose a request, manage the byte stream that contains the response, and translate the headers and body of the response into useful in-memory objects. Furthermore, *any* type of parallel code execution – not just in the context of S3 reads – incurs overhead: worker spawning, context switching, interworker communication, and memory management (of both main and CPU cache memory).[28] Individual read operations are computationally inexpensive; there are few computing platforms we would be likely to consider for scientific data processing that would be meaningfully stressed by S3 reads in series. However, executing and handling multiple read operations in parallel can quickly become burdensome.

More specifically, effective worker count is often bottlenecked by the number of available CPU threads. Single-thread performance is not typically very important for retrieval (although it can provide some improvements if inline decompression is involved). Other factors like CPU cache size and speed can certainly be important, depending in part on the size of the retrieved subsets, but they are not generally dominant.

---

[26] Eventually S3 will begin throttling you, or you may even overwhelm intermediate network layers.
[27] Because parallelism can use a lot of CPU, long-running parallel operations on CPU-metered 'burstable' EC2 instances like the t3/t3a/t4g families will tend to increase EC2 instance charges, often in an unpredictable way. An instance type without CPU metering will often be more economical in these applications, even if its base price per hour is higher.
[28] Because S3 read operations are not very computationally complex, we assume here that worker handling is managed gracefully enough in our technology stack that we can simply abstract these overhead costs into overall system resource pressure. This is *not* the case in every stack, however, so take care.

Note that parallelism also incurs working memory costs – at minimum, additional bytes must be held in memory as transfers complete, and handling and concatenation of array elements generally also has memory costs. Peak memory usage usually scales at least linearly with the number of workers. For small subsets or small numbers of target objects, this often doesn't matter, but in some cases, it restricts the feasibility of coarser subsetting strategies, mandates the use of (generally more expensive) servers with more working memory, or requires implementation of careful (and usually slow) optimizations like scratch-writes to local storage or aggressive inline garbage collection.

All this being said, **S3 read operation parallelism is also often limited by available bandwidth or request rate**. If you saturate the total transfer volume your network interface or intermediate carriers can handle, or hit a number of requests/second at which S3 (or an intermediate carrier) begins aggressively throttling your requests, further increasing your worker count is useless. These network limitations are highly environment-dependent; you are more likely to find that they are bottlenecks from your living room than from an EC2 instance. They are also application-dependent: if you are taking full-extension subsets from large FITS objects, you are very likely to find that your useful worker count is limited by bandwidth even on EC2.

tradespace: worker count and time

There are three primary domains of the worker count <-> time tradespace for S3 reads:

domain 1

At low worker count, adding workers ~linearly increases performance. In this domain, the ratio of parallel execution time to serial execution time is roughly equal to the integral of 1/n^2 evaluated from 1 to worker count, or, more succinctly, 1 - (worker count - 1) / worker count. (Sometimes there is also a constant offset (which can be positive or negative) from strict serial execution.) This means that the marginal time savings provided by each additional worker drops off quite quickly even within this domain: if an activity takes 20 seconds in series, adding one worker saves 10 seconds, adding another saves another 3.3 seconds; adding another saves another 1.7 seconds, and so on.

domain 2

Eventually, a variety of factors push operation rate increase out of this linear domain; adding additional workers after this has diminishing performance returns per worker (unless you hit the "wall" of domain 3 first). This decrease is multifactorial; it may be driven by parallelization overhead, request throttling, transient contact with bandwidth limits, etc., in any combination. Because the rate of decrease is so environment- and application-dependent, there is no good way to produce a general-purpose formula for it; if you are looking to optimize worker count in this domain, you must benchmark it in the context of your use case and execution environment. But here are a couple of heuristics:

Unless some other factor takes over first, applications tend to cross into domain 2 at ~2 workers per thread, and diminishing returns tend to be sharp after ~4 workers per thread. Past this, cache thrashing and excessive context switches start to become very important. If computationally-expensive intermediate steps like inline decompression are involved, this dropoff is considerably sharper.

For smallish cutouts retrieved by EC2 instances from S3 buckets in the same region using full-line or similar subsetting techniques, a transition into domain 2 generally becomes quite noticeable by 12-16 workers – even on instances with many more available threads – and very often earlier. This is most likely due to request throttling.

domain 3

There is a "wall" past which higher worker count never helps and usually hurts. You run out of threads or bandwidth, or reach a point where parallelization overhead costs more than the benefit offered by an additional worker, or S3 stops tolerating your flood of requests.

In any given application or environment, the factor(s) that produce this wall are *not* necessarily the same ones that degrade marginal performance increase per worker in domain 2. Sometimes this even means you never reach domain 2: for instance, sometimes 2 workers are 2x as fast as 1, and 3 workers are 3x as fast, and then suddenly you run out of bandwidth and adding a 4th worker just uses more of your CPU resources for no benefit.

Conversely, sometimes the factor(s) that drive the decrease in per-worker returns *are* the same as those that produce this wall. In these cases, domain 2 smoothly transitions into domain 3 as the performance/worker curve flattens to horizontal.

In some very bad cases for parallelism, you never even reach domain 1: for instance, if you are transferring large objects on a weak wifi connection, one worker may eat up all your bandwidth.

enabling technologies

S3 interfaces

Because of the development time costs of manually composing S3 API requests, it is usually preferable to use an off-the-shelf interface to the S3 API. In this investigation, we principally explored two types of S3 API interfaces: S3 FUSE implementations (principally *goofys*) and Python-language S3 support libraries (principally *fsspec/S3FS*). However, there are *many* other S3 interfaces which might be worth considering for other use cases. Some are software libraries designed for use in other code (including Amazon's official libraries like *boto3* for Python and the

AWS SDK for Java, and unofficial libraries like *rusty-s3* for Rust). Others are self-contained applications (like the AWS CLI), or components of more general-purpose applications (like Cyberduck's S3 features). Many of these interfaces, however, do not provide abstractions for subsetting. For instance, while it is possible to perform subsetting operations with the *boto3* Python library, you must do so by explicitly passing byte ranges to *botocore.client.S3.get_object()* calls. This is not very convenient.

FUSE

FUSE (Filesystem in USEr Space) is a framework that allows userspace applications to access external data volumes or data sources as if they were 'normal' filesystems, without requiring elevated privileges. It was originally developed for Unix-like systems, and ships with most widely-used Unix variants, including most major Linux distributions; versions also exist for MacOS, and a compatibility layer exists for Windows.

FUSE implementations exist for a wide variety of filesystems, services, and protocols, from exFAT to FTP to S3. FUSE interfaces to S3 essentially 'trick' the operating system into seeing S3 objects as files. Filesystems have existed in something close to their current form for almost half a century, and almost all software knows how to work with files. **The filesystem is the ultimate compatibility layer. This is the primary advantage of FUSE interfaces to S3.**

This means that FUSE allows scientists to use their existing workflows in the cloud. Most applications designed for local FITS files work transparently on FUSE-mapped S3 objects. For instance, when combined with FUSE, STScI's *fitscut* tool instantly becomes an S3 subsetting application.[29] GUI applications like *ds9* and *fv* offer browsing interfaces to FITS objects in S3. This applies to general-purpose system utilities as well: for instance, object management is made easier by the fact that GUI file managers work on FUSE-mapped objects, as do shell commands like *ls*, *cp*, and *rm*. Operating system-level features like memory caching also work with FUSE interfaces, meaning that you get many subsetting optimizations for 'free'.

Similarly, FUSE interfaces are compatible with scientific codes written in almost any computer language. Whether an astronomer works in MATLAB, C, Python, Rust, R, IDL, Go, shell script, or Fortran, a FUSE interface will generally allow their codes to work with S3 objects.

other advantages of FUSE-backed S3 access

- FUSE is mature (in continuous release since the late 1990s), well-supported, and extremely widely-used (for instance, Android manages SD card mounts with FUSE, meaning that a large proportion of all living humans use FUSE daily).
- it 'just works'. Astronomers who are not software engineers can simply 'mount' an S3 bucket, pretend the objects in the bucket are files, and suffer no serious consequences for

[29] Is it fully optimized? Probably not. But initial testing suggests that it is satisfyingly performant for S3 subsetting with absolutely no additional development work.

doing so. Although a particular code may not be fully *optimal* for use with file-mapped S3 objects, it can still leverage many of the advantages of cloud storage without the project bottlenecks and costs sometimes associated with cloud development. **This vastly lowers the learning curve for scientists who wish to access cloud resources.**
- Because they can leverage many existing built-in optimizations for file access, codes that run against S3 FUSE interfaces are often highly performant out-of-the-box without additional development work.

disadvantages of FUSE-backed S3 access

- An S3 FUSE interface must be installed on the host – while *goofys* and similar interfaces provide compiled binaries, making installation very easy, this may still present barriers in some environments.
- FUSE kernel libraries (*libfuse* or equivalent) must be installed on the host; they ship with many operating systems, but not with all, and this may add installation complexity.
- While most environments permit non-privileged users to manage FUSE mounts (this is part of the point of FUSE), this presents security issues in some computing frameworks (like Kubernetes). In these environments, FUSE mounts must be managed by system administrators, which reduces flexibility and may increase staff workloads.
- It is generally difficult to perform flexible optimizations on the per-operation level – the interface abstracts too much. While application-appropriate settings can often be defined at mount time, there may be cases in which greater control is desired.
- The ways objects differ from files make certain operations inefficient in ways no amount of trickery can fix. Read operations are generally fairly safe; it is much more common to encounter bad cases in write operations.[30]
- While FUSE interfaces reduce complexity in many ways, they *do* add an additional layer to the technology stack, and, concomitantly, an additional point of failure.

language-specific libraries

Interfaces to S3 can also be implemented at the software library level rather than the OS (or application) level. Our type example in this investigation was the Python interface *S3FS*, an implementation of the Python *fsspec* framework for building interfaces between external data sources and Python filelike objects. As part of the Fornax effort, G. Barentsen developed an extension to the widely-used Python *astropy* library's FITS interface (*astropy.io.fits*) that utilizes *S3FS* to allow *astropy* to fluidly access FITS resources stored in S3; it includes subsetting features[31]. *astropy* is a linchpin of the contemporary astronomical software ecosystem and, as such, something of a special case, but we can make several general observations about software library S3 interfaces as opposed to FUSE-backed application/OS layers:

---

[30] For instance, it is impossible to perform random writes to objects, so applications that attempt random writes will in fact delete and recreate the entire object over and over. Other kinds of interfaces won't make this work well, either – it's impossible – but they often won't even let you *try*.
[31] Accessed via the *section* attribute of *astropy.io.fits.hdu.image.ImageHDU*.

- Without additional software development to write wrappers or leverage foreign function interfaces, **they can only support codes written in their language. For instance, *S3FS* is of no use to an astronomer working in MATLAB. Similarly, they cannot support external applications without glue or wrapper code**.
- Optimizing codes backed by libraries of this type typically takes **more effort than optimizing codes backed by FUSE layers**, because they are unable to leverage OS resources as readily.
- However, they are often easier to optimize for specific use cases, particularly for developers who are highly proficient in the language and frameworks they support (although optimization, of course, comes at the cost of additional development time).
- Similarly, they can be integrated into compatible software libraries to extend their capabilities to cloud, which can be especially useful for widely-used support libraries.
- **Unlike application layers, software libraries of this type often do not require installation of OS-level libraries.** This sometimes makes them more environment-agnostic.
- **Conversely, they do require installation of language libraries.** In Python, environment management tools like *conda* make this relatively easy, but there are nevertheless pitfalls. It is much more difficult in some languages. Furthermore, compatibility between language libraries and system libraries is never quite guaranteed. To some extent, this pushes compatibility problems onto the language interpreter, environment, and/or runtime.

tile compression

Many existing astronomical data sets are very large. Managing this by storing astronomical data in compressed formats has been common for decades, although storing data in massive sets of uncompressed raster arrays remains *more* common. However, due to many factors, new missions are generating ever-greater volumes of data. These data volumes are increasing *much* faster than generally-available bandwidth is increasing, and somewhat faster than storage costs are decreasing. For these reasons, **it is increasingly urgent that data producers and archives have access to effective, usable compression schemes in order to keep their data manageable and accessible**.

Traditional compression modalities are *monolithic*: they compress entire data resources at once. For FITS, the most common technique is gzipping entire files[32]. This is extremely easy to implement and often fairly storage-efficient, especially on files whose volume mostly consists of sparse arrays. The major downside of this technique is that a monolithically-gzipped resource can only be interpreted as a stream; there is no consistent way to "seek" into it and find the specific data you're looking for. In the general case, **it is impossible to use subsetting techniques on monolithically-compressed files.**

[32] In other words, compressing them using one of the many implementations of the DEFLATE algorithm: zlib, libdeflate, ISA-L igzip, etc.

However, it is possible to perform compression on FITS files at a more granular level. Individual extensions can be compressed (although it is not particularly common to do so) which permits subsetting at the single-HDU level. But, for images, there is an even better technique: *tile compression*. Tile compression defines a tiling on an array[33], individually compresses each tile, and stores the compressed tiles in a binary table HDU (along with references to their location in the uncompressed array). **It is easy to subset tile-compressed arrays** – range requests can simply fetch the necessary row or rows of the table, and the tiles can be decompressed into array elements after retrieval. The FITS format supports tile compression with a variety of algorithms; we have found that the RICE algorithm is typically the most performant algorithm for subsetting (and most other purposes).[34]

Subsetting is sometimes faster and sometimes slower on tile-compressed images. Compression increases the transfer volume savings of each read operation but introduces decompression overhead. It becomes more time-efficient when transfer savings outweigh handling costs of inline decompression, **so the less transfer speed you have, the more time-efficient subsetting tile-compressed resources becomes.** Its time performance frequently ends up being a wash, or a tradeoff between CPU and bandwidth. It is *always* more transfer-volume efficient (and thus cheaper in terms of data egress) except in pathological cases. **But the real benefit of tile compression is *qualitative*: tile compression lets you subset compressed images, which is otherwise impossible.** Data storage and transfer volume can be enormously reduced without forcing users to perform full-object transfers for every operation. We believe that **tile compression is an extremely important enabling technology for cloud access to rapidly-growing astronomical data archives.**

Most FITS-reading and -writing libraries support tile compression, but their degree of support varies. One notable hiccup for Python users whose workflows rely on the rich *astropy* ecosystem is that ***astropy* does not support reading individual tiles from compressed images, whether from S3 or local storage.** This means that, unless and until *astropy* adds this feature, Python users who wish to subset tile-compressed FITS resources past the single-HDU level must use the CFITSIO-backed Python library *fitsio* or wrap non-Python tools like STScI's *fitscut*.

---

[33] For 2D arrays, it is common to simply treat each row of the array as a separate tile. There does not appear to be a standard convention for tile sizes on >2D arrays.

[34] It has certain downsides, notably that it is slightly lossy on floating-point arrays (although lossless for integer arrays). Its degree of lossiness is tunable, and we believe that it is likely to introduce only trivial amounts of error into most analysis pipelines. See our separate report on this topic: https://github.com/MillionConcepts/fornax-documents/blob/main/compression%20comparison.pdf

references

Amazon Web Services (AWS). *Amazon Simple Storage Service: API Reference.* API version 2006-03-01, primary release 2019-03-27, revision 2022-02-18. https://docs.aws.amazon.com/AmazonS3/latest/API/s3-api.pdf *(note: stable URLs to historical versions of AWS documentation are not available, so contents may change).*

Cheung, Ka-Hing, et al. *goofys.* 2015-2022. https://github.com/kahing/goofys

Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110. June 2022. https://www.rfc-editor.org/info/rfc9110.

Greisen, E. W. and Harten, R. H. "An Extension of FITS for Groups of Small Arrays of Data", *Astronomy and Astrophysics Supplement Series* 44, p. 371. 1981.

International Astronomical Union FITS Working Group (IAUFWG). *Definition of the Flexible Image Transport System (FITS).* Version 4.0 (language-edited update). 2018 August 13. https://fits.gsfc.nasa.gov/standard40/fits_standard40aa-le.pdf

NASA Office of Science and Technology (NOST). *Definition of the Flexible Image Transport System (FITS).* Version 1.0 (NOST 100-1.0). 1993 June 18. https://fits.gsfc.nasa.gov/standard10/fits_standard10.pdf

White, R. et al. "Tiled Image Convention for Storing Compressed Images in FITS Binary Tables." Version 2.3. 2013 July 2. https://fits.gsfc.nasa.gov/registry/tilecompression/tilecompression2.3.pdf