The Case for Dual-access File Systems over Object Storage

Kunal Lillaney[1], Vasily Tarasov[2], David Pease[2], Randal Burns[1]
[1]*Johns Hopkins University,* [2]*IBM Research—Almaden*

Abstract

Object storage has emerged as a low-cost and hyper-scalable alternative to distributed file systems. However, interface incompatibilities and performance limitations often compel users to either transfer data between a file system and object storage or use inefficient file connectors over object stores. The result is growing storage sprawl, unacceptably low performance, and an increase in associated storage costs. One promising solution to this problem is providing *dual access*, the ability to transparently read and write the same data through both file system interfaces and object storage APIs. In this position paper we argue that there is a need for dual-access file systems over object storage, and examine some representative use cases which benefit from such systems. Based on our conversations with end users, we discuss features which we believe are essential or desirable in a dual-access *object storage file system* (OSFS). Further, we design and implement an early prototype of Agni[1], an efficient dual-access OSFS which overcomes the shortcomings of existing approaches. Our preliminary experiments demonstrate that for some representative workloads Agni can improve performance by 20%–60% compared to either S3FS, a popular OSFS, or the prevalent approach of manually copying data between different storage systems.

1 Introduction

Some estimates predict that by 2020 the world will produce 44 zetabytes of unstructured data each year [75]. Users and enterprises increasingly look to object storage as an economical and scalable solution for storing this unstructured data [42,44]. In addition, object stores can offer simplified management, ease of use, and better data durability than traditional file systems [34, 35, 59]. These factors have led to the emergence of a rich ecosystem of object stores, both in the cloud and on-premises [2, 11, 16, 19, 21–23]. It is predicted that in 2019 more than 30% of the storage capacity in data centers will be provided by object storage [38] and by 2023 the object storage market is expected to grow to $6 billion [40]. Object stores are increasingly being used for tasks that have traditionally required file systems, such as data analytics, media streaming, static web content serving, and data archiving [63, 83].

Object store and file system interfaces have fundamental *namespace* and *data access* differences. Object stores are typically characterized by RESTful access, flat namespaces, immutable data, relaxed consistency, and rich user-defined metadata. These limited object storage semantics make them unsuitable for some applications. For example, object stores do not support partial writes, so any update must rewrite an entire object. As a result, existing workflows that rely on partial writes need to be redesigned for object storage in order to avoid the performance penalty of rewrites [77]. Object storage systems also do not provide a hierarchical namespace. Many applications, such as genomic workflows [54, 61], depend on a namespace hierarchy, file renames, and file pointers, none of which are supported by object stores.

Users and infrastructure providers have adopted different approaches to overcoming the disparity between files and objects, especially for recently emerging use cases. In one approach, distributed file systems are paired with object storage, and data is periodically copied over to a file system from the object store for data analytics [4, 13, 27]. This approach offers the full spectrum of file interfaces but is unwieldy for large datasets and can lead to over-provisioning, cost inflation, reduced backup efficiency, additional time for data transfer, and poor quality of service (QoS) due to storage sprawl [71]. An alternate approach, which we term *object storage file system* (OSFS), overlays file interfaces on top of existing object stores [41, 49, 60, 62, 64, 80]. This approach gives a single location for data and provides ease of administration and use. Given its benefits, we focus on OSFS in this paper.

Based on our exhaustive survey we believe that existing OSFSs fall short of simultaneously providing two capabilities that are essential in modern multi-application infrastructures: (i) dual access to data and (ii) an efficient file system interface. We define dual access as the user's ability to directly address data through a native object or file interface without going through an intermediary layer or having to look up the file-to-object mapping. Quintessentially, dual access should be *intuitive*, the same data should be accessible by the same name thorough either interface. An example of a workflow that benefits from dual access is a movie file being accessed by a transcoder using `fread`, then being made available by a Content Distribution Network (CDN) service via `GET`.

In this paper, we first present a class of applications that could potentially benefit from an OSFS with a dual-access capability. We further identify requisites for designing such a system so that it has a high probability of adoption. We then briefly describe a potential solution, called Agni, with preliminary insights into its design. Finally, we provide some elementary performance comparisons between Agni and S3FS, a popular OSFS for the discussed use cases, as well as comparing the prevailing manual data transfer approach.

[1] The name of our system is inspired by *Agni*, the two-faced Hindu deity. Just as the deity has two faces, our storage system has two interfaces.

2 Use cases

Based on our interactions with real customers we present two motivational use cases for a dual-access storage system. We believe that these examples represent a growing class of applications which can greatly benefit from dual access.

Media transcoding, editing and analytics. The vast majority of media-based workloads are characterized by the generation of numerous media files, followed by some form of data processing on those files, and finally relocation to an archive for cold storage or dissemination to end users. Data processing can be in the form of video editing, trimming, applying cinematic effects, overlaying different audio channels, and transcoding into different media formats. Most of the software involved in the processing stage relies on file interfaces. At the same time, cost effectiveness and near-limitless capacity drive the Media and Entertainment industry towards large-scale adoption of object storage [35, 43].

A simple example workflow entails the generation of raw video footage from film studios which is directly moved into object storage using object interfaces. That raw footage is then edited and transcoded into different media formats and resolutions to support compatibility with different platforms. When processing is complete, both the original raw videos and the different transcoded formats are moved back into object storage. Data intended for end-user consumption is delivered directly from object storage using CDNs.

Gene alignment. Currently scientists are struggling with massive data sets (petabytes and growing) in the domain of human genomics [58]. The challenge here is to effectively store large amounts of data, conduct genome analysis at scale, and eventually share this data among the research community. Object storage is well suited for solving these issues and serving as the central data repository for all genomic data [35]. However, most of the current genome analysis pipelines rely on file system capabilities for analysis [39, 54, 57, 61, 68]. A typical example of a gene sequencing pipeline involves generation of human genome sequences at laboratories. This data is then moved to a central data archive for durability and dissemination. The data is downloaded over the network to a local file system or network-attached device for gene sequence alignment. Different researchers run their own versions of sequence aligners over the unaligned data with tools such as Bowtie2 [39]. Sequence-aligned data is re-uploaded to the central archive (based, e.g., on object storage) and shared with other researchers. Scientists re-download the data and analyze it using tools such as SAMtools [55] and BCFtools [76].

Use cases in other domains such as neuroscience [48, 53, 56], geoinformatics [70, 72], machine learning [52], industrial process control [47], and computer vision [45, 65] can also benefit from dual access.

3 Design requirements

We identify the design requirements for a system which would be effective for the discussed use cases. These are based on our analysis of use cases, interactions with domain experts, and prior experience in developing storage systems. We present them in descending order of priority.

Object based. The system should employ object storage to store its data. This allows the system to realize the benefits of scalable, low-cost object stores. While it is technically easier to provide object interfaces over file systems, such a design choice does not inherit the desirable properties of object stores. Furthermore, users and enterprises are already entrenched in object storage and thus an object-based system would require minimal changes to either the vendor or storage technology.

Intuitive dual access. To address the storage sprawl issue the system should provide *intuitive* dual access to the data. We stress being intuitive because this is a vital property for high usability and wide adoption. The most intuitive approach to dual access is to adopt a 1⇒1 mapping. For example, the contents of a file should be fully stored in and accessible through a single object. We do not consider a system which splits a file into multiple objects with either 1⇒N or N⇒N mappings as one that provides intuitive dual access.

Generic and backend-agnostic. The system should be deployable over standard APIs, which is important for high flexibility, avoiding vendor lock-in, and the ability to use object stores that are already deployed in customer environments. We assume a limited, yet representative, set of operations: (i) `PUT(name, data)` adds a named object; (ii) `GET(name, range)` fetches a specific range from the object or the entire object[2]; (iii) `DEL(name)` deletes an object by name; (iv) `MULTI-PART DOWNLOAD(name)` retrieves an object in parallel[3]; (v) `MULTI-PART UPLOAD(name, data)` adds a named object in parallel. Based on our experience, these are a common subset of object interfaces available across all popular object stores.

Efficient file interface. In-place updates and metadata operations form an integral part of numerous application workloads [53, 82]. Given the disparities between object and file interfaces, providing both dual access and good performance is challenging. Existing OSFSs support either intuitive dual access or good performance for specific file interfaces, *but not both* (Section 4). For instance, a simple 1⇒1 mapping that maintains a file in a single object offers the most intuitive dual access from the user's perspective, and many existing dual-access OSFSs implement this mapping [10, 14, 32, 41, 51, 62, 64]. However, 1⇒1 mapping can drastically reduce the performance of partial file writes and file renames. This is because even a small modification to a

[2] A ranged `GET` is also referred to as a partial `GET`

[3] Large objects can be broken into chunks and downloaded/uploaded in parallel. A chunk is referred to as a `PART`.

file or the renaming of a file requires a `GET` and a `PUT` of an entire object. As a result, OSFSs either do not support partial writes at all or incur significant performance overhead. Other systems sacrifice intuitive dual access and adopt 1⇒N, N⇒1, or N⇒N file-to-object mapping policies [24, 30, 79, 80].

Coherent namespaces. Data is accessed through both interfaces, and the system should support coherent object and file namespaces to enable intuitive access. Object storage features a flat namespace with key-value style semantics. An OSFS could use object names with embedded path components to mimic a directory structure for the user's convenience, considering '/' in the object names as delimiters. But this can lead to various incoherencies in the file system namespace during creation and deletion of objects. For example, assume that there exists a directory named `/A/` in the file system namespace; it would lead to an incoherence if the user were to `PUT` an object named `A` in the object namespace. These incoherences only get compounded when we start to add support for directories, hard links, and symlinks. Moreover, this could lead to data either becoming unreachable or accidentally overwritten. The current set of systems rely on the user's cognizance to prevent any incoherence, and do not take this issue into consideration.

Eventual data access consistency. The system should ideally ensure that data access is consistent across both object and file interfaces. We closely examine the use cases and draw the conclusion that though data is accessed through both interfaces, rarely is this concurrent. In our view, this property should be exploited for performance improvement when designing the system. We further believe that eventual consistency is a suitable model, especially given that object storage users and applications already expect it [37, 73].

Distributed. Object storage is inherently scalable and only a *distributed* file system can fully utilize this characteristic. The massive datasets of the presented use cases also indicate that a solution would need to scale out beyond the compute power of a single node. One approach to this is to implement a full-fledged distributed file system that performs byte-range locking. However, on closer examination of the use cases, the datasets would need distributed access, but not necessarily to the same data. A potential file system would probably be able to operate with weak cache consistency, since individual nodes would likely work on distinct parts of the data. Some cloud providers even offer NFS over object storage [7, 31]. We believe that a distributed file system with NFS V3-like semantics, but not weighed down by the scalability and performance limitations of NFS, would provide a good balance between consistency and performance.

Unified access control. The differences between file systems and object storage extend to access control methods. Access control in object storage is more detailed than the Access Control Lists (ACLs) that are available on standard file systems, and the two tend to be incongruent. Current OSFSs do not provide unified access control between files and objects, i.e., the permissions set for an object are not reflected in its file equivalent or vice-versa. However, enterprises desire this unification across interfaces. In addition, there is a vast diversity in the nature of access control supported by different object storage vendors, which presents a dilemma for a backend-agnostic system. For example, some object stores support object level permissions [1, 8, 17] while others only offer permissions at the container level [18, 26].

4 Existing systems

We surveyed existing systems and classified them into two broad categories, **I** and **II**, based on the approaches presented in Section 1. We further subdivided **II** into three types, **IIa**, **IIb**, and **IIc**, based on their support for dual access and efficient file interfaces. The results of our survey are summarized in Table 1, illustrating that despite a fairly high number of existing relevant systems, none of them meet all of the desirable requirements. (We do not include file systems which were designed to use object-based storage devices (OSDs), such as Lustre [69], or systems like SwiftOnFile [33], which is an object store deployed over a distributed file system, because of their prohibitive limitations on dual access and generality.)

I : File systems paired with object storage (in the table). These are hierarchical storage management systems based on block and object storage. A distributed file system is deployed over block storage and paired with object storage. Data is either transparently or manually migrated between them. This approach is inefficient and cumbersome for large datasets and leads to storage sprawl with the continuous transfer of data between different storage tiers.

II : OSFSs operate over existing object storage systems. (a) **Inefficient file system interface.** These systems offer intuitive dual access but support only simple or inefficient file interfaces. They can be viewed as object stores with rudimentary file system properties. (b) **No dual access.** These systems offer the full set of efficient file interfaces but do not support intuitive dual access. They are essentially distributed file systems with few object-like features [4]. An example is CephFS, which is neither generic nor does it support dual access. It only operates over RADOS, its own object store, and though it does store data in RADOS, this data cannot be accessed via object interfaces. (c) **Best of both worlds.** These systems offer both intuitive dual access and the full spectrum of efficient file interfaces. ProxyFS and OIO-FS fall in this category. However, they are neither generic, since they operate only on specific object stores, nor distributed. Moreover, they provide only limited object interfaces or support only reads. ProxyFS adopts N⇒N object-to-file mapping and relies on Swift's ability to present a scattered object as a single entity via the object interface. Swift containers (buckets)

[4] We do not mention proprietary systems like Cloudian HyperFile [12] and MagFS [20] because there is little public information about their internals. To our knowledge, these systems can also be characterized under this category.

Table 1: Characterization of existing object storage based file systems.

Name	Dual access	Gen-eric	File-system interface support and efficiency	Name space	Distr-ibuted	Access control
Goofys [41], SVFS [32], Riofs [51]	✔	✔ [a]	No support for partial writes, renames and metadata	✘	✘	✘
S3FS [64], Blobfuse [10], Gcsfuse [14]			Incomplete support. Inefficient file and metadata updates		✔ [b]	
YAS3FS [62]						
MarFS [49], SCFS [36]	✘	✔ [a]	No support for partial writes	✘	✘	✘
S3QL [30]			Full support. Optimal file and metadata updates		✔	
ObjFS [24], Cumulus [79], BlueSky [80]						
CephFS [81]		✘				
Elastifile [13], CloudFS [27], FSX [4]	✘	✔	Full support. Optimal file and metadata updates	✘	✔	✘
OIO-FS [25]	Limited	✘	Incomplete support. Optimal file and metadata	✘	✘	✘
ProxyFS [28]				Future [c]		
Agni	✔	✔	Full support. Optimal file and metadata updates	Future [c]	✔ [b]	Future [c]

[a] Currently supports a single object store but uses only generic object operations. We consider it generic because it could be ported to other object stores.
[b] Currently support NFSv3 semantics. Consistent cache coherency can be achieved with the use of distributed locking mechanisms which are currently absent. This is distinct from running NFS or equivalent services over existing object storage.
[c] This is being actively discussed and future versions are expected to support it.

need to be pre-configured for dual-access operation and this feature cannot be enabled retroactively. ProxyFS is also the only other existing system which considers the issue of maintaining a consistent namespace, and is attempting to tackle it. It relies on the internal capability of pre-configured Swift containers to maintain coherence but this is currently limited to only a few cases [29]. OIO-FS adopts a 1⇒1 object-to-file mapping and supports dual access only for reads.

We conclude that none of the existing systems fulfill the desired feature set from Section 3. This prompted us to design and develop Agni ( ) to overcome such limitations.

5 Agni

Agni is an efficient dual-access OSFS that provides eventual consistency through both file and object interfaces while utilizing only commonly implemented object APIs and capabilities. The effect of an operation through either interface is eventually reflected through the other interface. It implements a multi-tier data structure that provides high performance for partial writes by accepting them in memory, persisting them to a log, and aggregating writes when merging the log. Logged updates are indexed temporally which allows the system to update existing files even while flushing data to the log. Agni merges logged updates asynchronously and presents a single object to the user. In addition, we have implemented useful dual-access user APIs that, among other functions, allow users to synchronize files with objects on demand.

Agni commits all file writes to a log that resides in object storage in order to avoid read-modify-write overhead for partial writes. The system maps a single file to a single object, but does so across three distinct storage tiers: (i) *cache*, (ii) *log*, and (iii) *base*. File-based applications read data from and write data to the top-most tier—the cache. Agni fetches data into the cache on demand. Asynchronously, the *flush* process writes data out from the cache to the intermediary tier—the log. Periodically, a *merge* process asynchronously reconciles data from the log with the object in the bottom-most tier—the base. The resting state of the system is when all data is located in the base. At times parts of file system data can reside in all three tiers, but flush and merge processes eventually return the system to its resting state. We call this approach eventual 1⇒1 mapping, and the associated delay *file-to-object visibility lag*. Figure 1 depicts the three tiers, and the relationship between them and applications. The file interface always sees consistent data while the object interface sees consistent data when there is no dirty data in the cache and the log.

The log provides a transient staging ground for partial writes. We defer the read-modify-write until the file is merged into the base tier. To ensure a faster and simpler merge, Agni maintains a separate log for each file. Updates made via the object interface are reflected to the file system using object notifications[5]. We refer to this time delay as *object-to-file visibility lag*. Weak cache consistency across distributed nodes is maintained via a publish/subscribe messaging service.

Currently Agni assumes the absence of concurrent file and object data updates, and does not implement any locking. This can lead to data corruption when concurrent data writes do occur. For example, the base can be overwritten because of a write through the object interface even when a file is open. In this case, data written through both interfaces could potentially be interspersed. However, we expect this to occur rarely in practice. In the future we plan to explore the use of checksums to guarantee object-to-file consistency to address this issue: every object in the log would be associated with a checksum of the base that would ensure unmerged updates

[5] Most object storage can be configured to notify applications for specific object events [3, 9, 15].

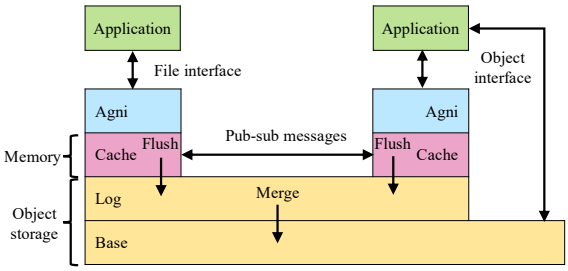

Figure 1: A design overview of the Agni architecture with different storage tiers and dual data access interfaces.

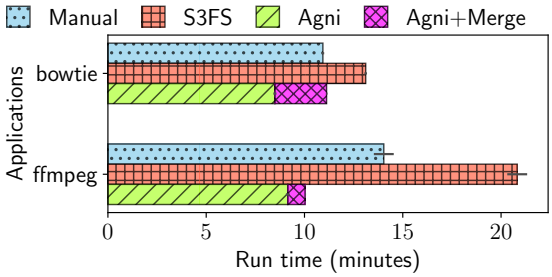

Figure 2: Comparing runtimes for existing applications.

are not merged with an overwritten base. In addition, we implement `coalesce()`—a file interface to force a synchronous merge. On its completion, updates from the file interface are reflected in the object interface.

We implemented Agni using FUSE [78]. Our system supports multiple file systems concurrently, one per object store bucket. For this paper, we store metadata in Redis [67] and use tmpfs as Agni's cache. Agni currently runs on AWS S3 [1], OpenStack Swift [26], and Google Cloud Storage [17]. The object notification processor uses the serverless compute service [5], and updates are conveyed to distributed nodes using a cloud native messaging service [6]. Currently Agni does not support a consistent namespace or an integrated access control mechanism. Our implementation is written in Python; it is open-source, and available at `https://github.com/objectfs/objectfs`.

Preliminary evaluation: We validate the performance of Agni for two different applications by comparing it to S3FS, a popular OSFS, and to a script which imitates manually copying data. We prioritized our comparison to these approaches because they satisfy two principal criteria: dual access and being backend-agnostic. Of the 19 other file systems in Table 1, only 7 systems qualify and only two of those have near-complete POSIX support—S3FS and YAS3FS. We choose to compare Agni to S3FS because of its popularity and maturity [64]. S3FS is implemented in C++ using FUSE, and uses tmpfs as its local cache. It can be mounted across multiple nodes, but there is no mechanism to maintain coordination between them and thus it is not a distributed file system.

Benchmarks are performed on Amazon Web Services (AWS) using an `m5.4xlarge` instance with 16 vCPUs, 64 GB of RAM, and *AWS high* network bandwidth. For both systems we use a 36 GB in-memory file system as a cache, and for Agni we allocate 4 GB to the Redis server.

We simulate the media use case by running *ffmpeg* [74], a transcoding application, to transcode 14 MPEG files (total 32 GB)[6] to MOV files. Gene sequence alignment is simulated using *bowtie2* [39], a gene sequence aligner, to align a single large genome file of 8 GB[7]. The manual data transfer workload attempts to mimic the case for both file systems paired with object storage and a manual user performing the actions. Figure 2 depicts the results. Agni performs better because it overlaps I/O and computation. It can upload data to object storage while parts of it are under processing. Other systems have to wait for the entire file to be processed before their upload can begin. The merge time denotes the file-to-object visibility lag after the data is uploaded. Agni performs 40%-60% faster in case of ffmpeg and 20%-40% faster in case of bowtie2 when compared to both of the existing approaches.

Future work: Agni is a system under active development, and lacks some attributes essential for its deployment in a production environment. Our current focus is to implement coherent namespaces and access control over our dual-access system. Towards this goal, we plan to support two operating modes for Agni: (i) Mode I will offer complete intuitive dual access and the best object interface performance, but will only support basic object interfaces, and applications will have not have the ability to verify the reflection of file interface updates through the object interface. Additionally, this mode will able to rectify namespace incoherencies that can be remedied retroactively without data loss, and will be able to identify the set of incoherencies that will require user intervention to resolve. Finally, access control in this mode will be dependent on the support offered natively by the object store. (ii) Mode II will actively prevent the occurance of namespace incoherence, and will support filesystem-like operations such as creation and deletion of directories, symlinks, and hard links via the object interface. Access control will be uniform and independent of the object store. In addition, applications will have the capability to verify the reflection of file interface updates before accessing an object. However, this mode will have lower object interface performance and intuitiveness.

Acknowledgements: This work was funded by grants from IBM Research, NSF (IOS-1707298, OCE-1633124, IIS-1447639) and NIH (1R01NS092474, 1U01NS090449). The authors would also like to thank Michael Mesnier (Shepherd), Peter Guglielmino (Media & Entertainment, IBM) and Christopher Wilks (Genomics Lab, JHU) for their valuable insights.

[6] Media files were downloaded from Internet Archive's Sci-Fi section [50].

[7] A human genome dataset [46] was downloaded from the Sequence Read Archive hosted at National Center for Biotechnology Information [66].

Discussion topics for HotStorage'19

We would like to get the storage community's feedback on the following points:

1. Many enterprises are looking to move towards multi-cloud or hybrid-cloud setups. We'd like to discuss the challenges of deploying dual-access storage systems in such organizations.
2. We leverage the acceptability of weak data consistency across both interfaces to improve Agni's performance. This works for our use cases, but are there use cases that require strong data consistency? If weak consistency is sufficient, what would be acceptable object-to-file and file-to-object visibility lags?
3. We describe a set of use cases in Section 2 which are well suited for dual-access systems. Are attendees aware of any other use cases (sourced from their real-world experience) which could benefit from such systems?
4. We have presented our rationale behind the design requirements in Section 3. Are there other design attributes which we have not discussed? Are there any aspects of our design which necessitate more deliberation or rethinking? Further, we believe that more data workflows are bound to migrate to cloud. Which additional design attributes would be important to better position Agni in cloud environments?
5. In Future work, we discuss our approach to tackling the unsolved challenges of presenting a coherent namespace and unified access control. The intuition is to provide support for a diversity of potential user requirements and choices of object storage. Does our approach appear to be feasible or are we overfitting? Are there other approaches to this problem and how do organizations currently address these issues?

References

[1] Amazon Simple Storage Service (S3). https://aws.amazon.com/s3/.

[2] Amazon Web Services Pricing. https://aws.amazon.com/pricing.

[3] AWS Event Notifications. https://docs.aws.amazon.com/AmazonS3/latest/dev/NotificationHowTo.html.

[4] AWS FSx. https://aws.amazon.com/fsx/.

[5] AWS Lambda. https://aws.amazon.com/lambda/.

[6] AWS SNS. https://aws.amazon.com/sns/.

[7] AWS Storage Gateway for Files. https://aws.amazon.com/storagegateway/file/.

[8] Azure Blob Storage. https://azure.microsoft.com/en-us/services/storage/blobs/.

[9] Blob Storage Events. https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blob-event-overview.

[10] BlobFuse. https://github.com/Azure/azure-storage-fuse.

[11] Caringo Swarm. https://www.caringo.com/products/swarm/.

[12] Cloudian Hyperfile. https://cloudian.com/products/hyperfile-nas-controller/.

[13] Elastifile Cloud File System. https://www.elastifile.com/product/.

[14] GCSFuse. https://github.com/GoogleCloudPlatform/gcsfuse.

[15] Google Change Notifications. https://cloud.google.com/storage/docs/object-change-notification.

[16] Google Cloud Platform Pricing. https://cloud.google.com/pricing.

[17] Google Cloud Storage. https://cloud.google.com/storage/.

[18] IBM Cloud Object Storage. https://www.ibm.com/cloud/object-storage.

[19] IBM Cloud Pricing. https://www.ibm.com/cloud/pricing.

[20] Maginatics MagFS. http://downloads.maginatics.com/MaginaticsMagFSTechnicalWhitepaper.pdf.

[21] Microsoft Azure Pricing. https://cloud.microsoft.com/en-us/pricing.

[22] Microsoft Azure Pricing. https://azure.microsoft.com/en-us/pricing/.

[23] Minio. https://www.minio.io/.

[24] ObjectiveFS. https://objectivefs.com/.

[25] OpenIO FS. https://docs.openio.io/18.04/source/arch-design/fs_overview.html.

[26] OpenStack Swift. https://docs.openstack.org/swift/latest/.

[27] Panzure CloudFS. https://panzura.com/technology/panzura-cloudfs/.

[28] ProxyFS. https://github.com/swiftstack/ProxyFS.

[29] ProxyFS Namespace. https://proxyfs.org/architecture/example-flows.html.

[30] S3QL. https://bitbucket.org/nikratio/s3ql/.

[31] Scality RING. https://www.scality.com/products/ring/.

[32] SVFS. https://github.com/ovh/svfs.

[33] SwiftOnFile. https://github.com/openstack/SwiftOnFile.

[34] A. Alkalay. Object storage benefits, myths and options. *IBM Blog*, Feb 2017.

[35] R. Bala, G. Landers, and J. McArthur. Critical Capabilities for Object Storage. Technical Report G00304492, Gartner, Jan 2018.

[36] A. Bessani, R. Mendes, T. Oliveira, N. Neves, M. Correia, M. Pasin, and P. Verissimo. SCFS: A Shared Cloud-backed File System. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, pages 169–180, Berkeley, CA, USA, 2014. USENIX Association.

[37] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a Database on S3. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 251–264, New York, NY, USA, 2008. ACM.

[38] A. Chandrasekaran, R. Bala, and G. Landers. Critical Capabilities for Object Storage. Technical Report G00271719, Gartner, Jan 2017.

[39] R. Charles, V. Antonescu, C. Wilks, and B. Langmead. Scaling read aligners to hundreds of threads on general-purpose processors. *Bioinformatics*, Oct 2018.

[40] S. Chavan, T. Banerji, and R. Nair. Cloud Object Storage Market Research Report - Global Forecast 2023. Technical Report WGR3496538, Wise Guy Reports, Oct 2018.

[41] K.-H. Cheung. Goofys. https://github.com/kahing/goofys.

[42] A. Cockroft, C. Hicks, and G. Orzell. Lessons Netflix learned from the AWS outage. *Netflix Techblog*, Apr 2011.

[43] T. M. Coughlin. Achieving A Faster, More Scalable Media Active Archive. Technical report, Coughlin Associates, July 2017.

[44] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good. The Cost of Doing Science on the Cloud: The Montage Example. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 50:1–50:12, Piscataway, NJ, USA, 2008. IEEE Press.

[45] A. R. Elias, N. Golubovic, C. Krintz, and R. Wolski. Where's The Bear?: Automating Wildlife Image Processing Using IoT and Edge Cloud Systems. In *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation*, IoTDI '17, pages 247–258, New York, NY, USA, 2017. ACM.

[46] M. Ferrer, S. J. C. Gosline, M. Stathis, X. Zhang, X. Guo, R. Guha, D. A. Ryman, M. R. Wallace, L. Kasch-Semenza, H. Hao, R. Ingersoll, D. Mohr, C. Thomas, S. Verma, J. Guinney, and J. O. Blakeley. Pharmacological and genomic profiling of neurofibromatosis type 1 plexiform neurofibroma-derived schwann cells. *Sci Data*, 5:180106, June 2018.

[47] E. Goldin, D. Feldman, G. Georgoulas, M. Castano, and G. Nikolakopoulos. Cloud computing for big data analytics in the Process Control Industry. In *2017 25th Mediterranean Conference on Control and Automation (MED)*, pages 1373–1378. IEEE, July 2017.

[48] K. J. Gorgolewski, F. Alfaro-Almagro, T. Auer, P. Bellec, M. Capotă, M. M. Chakravarty, N. W. Churchill, A. L. Cohen, R. C. Craddock, G. A. Devenyi, et al. BIDS apps: Improving ease of use, accessibility, and reproducibility of neuroimaging data analysis methods. volume 13, page e1005209. Public Library of Science, Mar 2017.

[49] J. T. Inman, W. F. Vining, G. W. Ransom, and G. A. Grider. MarFS, a Near-POSIX Interface to Cloud Objects. *;Login*, 42(LA-UR–16-28720; LA-UR–16-28952), Jan 2017.

[50] Internet Archive. https://archive.org/search.php?query=subject%3A%22Sci-Fi%22. Accessed: 2019-01-29.

[51] P. Jonkins. RioFS. https://github.com/skoobe/riofs.

[52] A. Kaplunovich and Y. Yesha. Cloud Big Data Decision Support System for Machine Learning on AWS: Analytics of Analytics. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 3508–3516, Dec 2017.

[53] G. Kiar, K. J. Gorgolewski, D. Kleissas, W. G. Roncal, B. Litt, B. Wandell, R. A. Poldrack, M. Wiener, R. J. Vogelstein, R. Burns, et al. Science In the Cloud (SIC): A use case in MRI Connectomics. volume 6, pages 1–10. Oxford University Press, May 2017.

[54] B. Langmead, K. D. Hansen, and J. T. Leek. Cloud-scale RNA-sequencing differential expression analysis with Myrna. *Genome biology*, 11(8):R83, Aug 2010.

[55] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, and R. Durbin. The Sequence Alignment/Map Format and SAMtools. *Bioinformatics*, 25(16):2078–2079, Aug 2009.

[56] K. Lillaney, D. Kleissas, A. Eusman, E. Perlman, W. Gray Roncal, J. T. Vogelstein, and R. Burns. Building NDStore Through Hierarchical Storage Management and Microservice Processing. In *2018 IEEE 14th International Conference on e-Science (e-Science)*, pages 223–233. IEEE, Oct 2018.

[57] K. Lim, G. Park, M. Choi, Y. Won, D. Kim, and H. Kim. Workload Characteristics of DNA Sequence Analysis: From Storage Systems' Perspective. In *Proceedings of the 6th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '14, pages 4:1–4:7, New York, NY, USA, 2014. ACM.

[58] V. Marx. Biology: The big challenges of big data. 498:255–260, 2013.

[59] E. Ottem. Don't Get Left Behind! Top Reasons You Need Object Storage. *Western Digital Blog*, June 2017.

[60] O. Ozeri, E. Ofer, and R. Kat. Keeping Deep Learning GPUs Well Fed Using Object Storage. In *Proceedings of the 11th ACM International Systems and Storage Conference*, SYSTOR '18, pages 128–128, New York, NY, USA, 2018. ACM.

[61] A. O'Driscoll, J. Daugelaite, and R. D. Sleator. 'Big data', Hadoop and cloud computing in genomics. volume 46, pages 774–781. Elsevier, 2013.

[62] D. Poccia. YAS3FS. https://github.com/danilop/yas3fs.

[63] A. Potnis. Worldwide File-Based Storage Forecast, 2018–2022: Storage by Deployment Location. Technical Report US44457018, International Data Corporation, Dec 2018.

[64] R. Rizun. S3Fuse. https://github.com/s3fs-fuse/s3fs-fuse.

[65] D. A. Rodriguez-Silva, L. Adkinson-Orellana, F. J. González-Castano, I. Armino-Franco, and D. González-Martinez. Video Surveillance Based on Cloud Storage. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, CLOUD '12, pages 991–992, Washington, DC, USA, 2012. IEEE Computer Society.

[66] Sage Bionetworks. https://www.ncbi.nlm.nih.gov/sra/?term=SRR6311433. Accessed: 2019-01-29.

[67] S. Sanfilippo and P. Noordhuis. Redis. http://redis.io.

[68] M. C. Schatz. CloudBurst. *Bioinformatics*, 25(11):1363–1369, June 2009.

[69] P. Schwan et al. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, volume 2003, pages 380–386, July 2003.

[70] Y. Shao, L. Di, Y. Bai, B. Guo, and J. Gong. Geoprocessing on the Amazon cloud computing platform — AWS. In *2012 First International Conference on Agro-Geoinformatics (Agro-Geoinformatics)*, pages 1–6, Aug 2012.

[71] H. Smith. *Data Center Storage: Cost-Effective Strategies, Implementation, and Management*. CRC Press, 2016.

[72] R. Sugumaran, J. Burnett, and A. Blinkmann. Big 3D Spatial Data Processing Using Cloud Computing Environment. In *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, BigSpatial '12, pages 20–22, New York, NY, USA, 2012. ACM.

[73] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based Service Level Agreements for Cloud Storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324, New York, NY, USA, 2013. ACM.

[74] S. Tomar. Converting Video Formats with FFmpeg. *Linux J.*, 2006(146):10–, June 2006.

[75] V. Turner, J. F. Gantz, D. Reinsel, and S. Minton. The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things. Technical report, 2014.

[76] C. Tyler-Smith, P. Danecek, R. Durbin, V. Narasimhan, Y. Xue, and A. Scally. BCFtools/RoH: a hidden Markov model approach for detecting autozygosity from next-generation sequencing data. *Bioinformatics*, 32(11):1749–1751, Jan 2016.

[77] K. Vahi, M. Rynge, G. Juve, R. Mayani, and E. Deelman. Rethinking Data Management for Big Data Scientific Workflows. In *2013 IEEE International Conference on Big Data*, pages 27–35, Oct 2013.

[78] B. K. R. Vangoor, V. Tarasov, and E. Zadok. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 59–72, Santa Clara, CA, 2017. USENIX Association.

[79] M. Vrable, S. Savage, and G. M. Voelker. Cumulus: Filesystem Backup to the Cloud. In *ACM Transactions on Storage (TOS)*, volume 5, pages 14:1–14:28, New York, NY, USA, Dec 2009. ACM.

[80] M. Vrable, S. Savage, and G. M. Voelker. BlueSky: A Cloud-backed File System for the Enterprise. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 19–19, Berkeley, CA, USA, 2012. USENIX Association.

[81] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[82] H. Xiao, Z. Li, E. Zhai, T. Xu, Y. Li, Y. Liu, Q. Zhang, and Y. Liu. Towards Web-based Delta Synchronization for Cloud Storage Services. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 155–168, Oakland, CA, 2018. USENIX Association.

[83] E. Yuen. Unlocking the Power of Analytics with an Always-on Data Lake. Technical report, Enterprise Strategy Group, Nov 2017.