Building NDStore through Hierarchical Storage Management and Microservice Processing

Kunal Lillaney*, Dean Kleissas†, Alexander Eusman*, Eric Perlman‡, William Gray Roncal†, Joshua T. Vogelstein§, Randal Burns*
*Dept. of Computer Science, †Applied Physics Laboratory, ‡Center for Imaging Science,
§Institute of Computational Medicine, Dept. of Biomedical Engineering
Johns Hopkins University
Baltimore, Maryland

***Abstract*—We describe NDStore, a scalable multi-hierarchical data storage deployment for spatial analysis of neuroscience data on the AWS cloud. The system design is inspired by the requirement to maintain high I/O throughput for workloads that build neural connectivity maps of the brain from peta-scale imaging data using computer vision algorithms. We store all our data on the AWS object store S3 to limit our deployment costs. S3 serves as our base-tier of storage. Redis, an in-memory key-value engine, is used as our caching tier. The data is dynamically moved between the different storage tiers based on user access. All programming interfaces to this system are RESTful web-services. We include a performance evaluation that shows that our production system provides good performance for a variety of workloads by combining the assets of multiple cloud services.**

***Index Terms*—Spatial Data, Big Data, Cloud Computing, Object Storage, Neuroscience**

I. Introduction

In 2013, we developed a scalable cluster called the Open Connectome Project [1] as a response to the scalability crisis faced by the neuroscience community. The system design was based on the principles of NoSQL scale-out and data-intensive computing architecture. This project currently has grown to 80 unique datasets that total more than 200 TB across different imaging modalities. It was deployed on the Data-Scope storage cluster [2] at Johns Hopkins University with MySQL and Cassandra as storage back-ends. We have recently migrated all Open Connectome Project data to a new cloud system, called NeuroData Store or NDStore, and continue to provide storage and analytics services to the neuroscience community.

Neuroscience has varied workloads that differ in scale, I/O bandwidth and latency requirements. One workload runs parallel computer vision algorithms at scale on high-performance compute clusters. This workload needs to scale, requires high I/O bandwidth, is not latency sensitive and performs large reads and writes [3]. One exemplar detects 19 million synapses using approximately 14,000 core hours in a 4 trillion pixel image volume [4]. Other workloads related to annotation of imaging data require low I/O bandwidth but are latency sensitive for small writes. Visualization presents a different workload that requires moderate I/O bandwidth and is very latency sensitive with small reads. Visualization platforms such as BigDataViewer [5] and NeuroGlancer [6] generate such workloads.

Our previous NoSQL architecture reached its scaling limit as neuroscience data evolved from terabytes to peta-bytes. In 2015, Intelligence Advanced Research Projects Activity (IARPA) announced the Machine Intelligence from Cortical Networks (MICrONS) program [7] to reverse-engineer the algorithms of the brain and revolutionize machine learning. This program will image multiple mouse brains, producing peta-bytes of data [8]. Relational databases on disk drives have difficulty providing the high IOPS required by this workload and are difficult to scale. Key-value stores, such as Cassandra, are designed to hold millions of key-value pairs and scale well. But, key-value stores that integrate solid-state storage and disk drives results in high operating costs for peta-bytes of data. Other considerations are the availability and reliability of data which are critical. Any data loss is catastrophic and inaccessible datasets can severely impede scientific discovery.

Currently, there is no single cloud service that provides peta-scale data with high I/O throughput. However, commercial clouds do offer many services to store data that include object stores, archival storage, memory clusters, relational databases, SSDs and key-value stores. To realize the reliability, availability, scaling on demand benefits of the cloud [9], we combine multiple services. One can use an object store to hold the data; e.g., S3 is scalable, cheap and reliable. But object stores are too slow for latency sensitive workloads, such as visualization, and not ideal for the random writes generated during annotation of imaging data. SSD's and memory clusters do offer low latency and work well with random writes but are cost-prohibitive at peta-scale.

Unfortunately, the overall experience of the scientific community with respect to the cloud has been mixed. Some open science projects have explored the idea of cloud computing for a wide spectrum of applications ranging from understanding Dark Matter [10], creating Montages [11], tracing DNA [12] to utilizing unused cycles on commercial clouds [13]. Mostly, the negatives have largely outweighed the positives with performance playing a major role [14]. Another demerit has been cost-effectiveness [15]. Several factors have led to this negative experience. Most of these explorations were conducted between 2008 and 2011 when there were far few service providers with more basic services. Today, commercial services such as Amazon Web Services (AWS) [16], Microsoft Cloud [17], Google Cloud Platform [18] and IBM Cloud [19] have evolved and offer a much mature set of services at competitive prices. Also, many of explorations tried to port their existing High Performance Computing (HPC) architectures to the cloud, rather than redesigning the software stack around cloud services. For example, Thakar [14] used SQL-Server to store data and managed SQL server instances manually. Since then, many managed databases services have emerged that would be easier and cheaper.

We have developed and deployed a hierarchical storage system on the cloud that replaces our NoSQL architecture. The system uses an object store as a base tier for its scalability and low cost and a memory cluster as a caching tier for low-latency I/O. We do not attempt to superimpose our earlier architecture on the cloud. Instead, we reinvent it to exploit cloud services to their full potential. Both, object stores and memory caches cannot be used in isolation because of their respective drawbacks. But they complement each other, negating the other's shortcomings. In addition, a memory-based fast-write buffer, called NDBlaze, accelerates random writes to the object store.

We decompose the application using micro-services for scalability, modularity and to avoid a single point of failure. Micro-services include server-less compute, distributed queues, and key-value stores, for ingest, data manipulation, and analysis. The use of these services aids in our quest to keep cost low while extracting the maximum performance possible for our different workloads.

We choose the AWS platform to develop and deploy our project. This choice is motivated by our prior experience with the platform. However, our design principles are not limited to this platform and could be implemented with other cloud providers.

II. Related Work

We draw our inspiration from hierarchical storage management systems that we adapt to a cloud environment. Many of the challenges we face parallel those that lead to the development of early hierarchical storage systems [20]. These systems were designed to use hard-drives in conjunction with other storage media, such as tapes. The multiple levels of storage are managed by the system so that the user sees a single storage address space. They have been widely used for serving videos [21], in file-systems [22] and data archives [23]. The ADSTAR Distributed Storage Manager (ADSM) [23] is one such system used for backup and archive. This system operated on multiple platforms and provided an illusion of infinite storage to the user. Another example is Coda [24], a file-system that ensured resiliency against server and network failures in large distributed systems. Coda descended from the Andrew File System (AFS) and allowed users to cache data locally to ensure constant data availability. Sprite [22] used client side caching for network file systems to increase I/O performance and reduce network traffic.

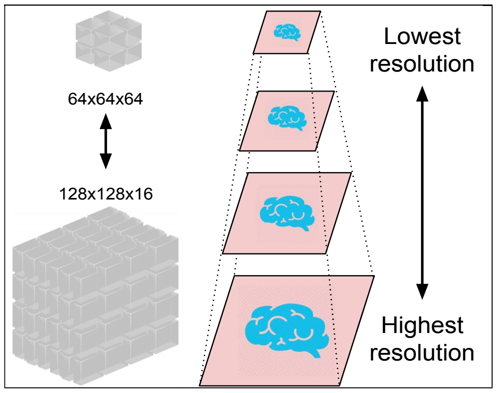

Fig. 1. The resolution hierarchy dictates the dimensions of the cuboids at each scale [1].

NDStore has a different usage pattern when compared with memory caching systems such as RAMCloud [25] and Memcached [26]. These memory caching systems were designed to act as caches over read-only data for applications in which data consistency is not an issue. They cannot ensure that the data will remain consistent when there are updates in the cache. We are not the first to propose the idea of hierarchical storage in the cloud. The idea of using caches over an object store has been used for a network edge cache [15]. However, the motivation behind their design was to alleviate the cost of object storage.

There have been numerous projects in the open science community which have attempted tera or peta-scale analysis in the cloud. CARMEN [27] is a platform developed on AWS for data-sharing and analysis for neuroscientists. The project moves computation close to the data and ensures faster analysis. It also uses scalable compute in the cloud for scaling up its workflows. CARMEN was designed to run data workflows which were not necessarily time sensitive or frequently updated. This system is incompatible with emerging workloads in neuroscience, such as visualization and manual annotation tools which require low latency I/O for practical use. Other projects [28] have implemented scientific pipelines on the Azure platform. However, none of their workflows are latency sensitive and utilize a single storage tier plus compute.

III. Data Design

The basic storage structure in our database is a dense multi-dimensional spatial array that is partitioned into rectangular sub-regions [1]. We call these subregions *"cuboids"* and they are similar to chunks in ArrayStore [29]. Figure 4 depicts a sample cuboid structure. Each cuboid is assigned an index using a Morton-order space filling curve. Space filling curves organize data recursively, so that any power-of-two aligned subregion is wholly continuous in the index [30]. They also minimize the number of discontiguous regions needed to retrieve a convex shape in a spatial database [31]. Morton indexes are easy to calculate and cube addresses are non-decreasing in each dimension so that the index works on sub-

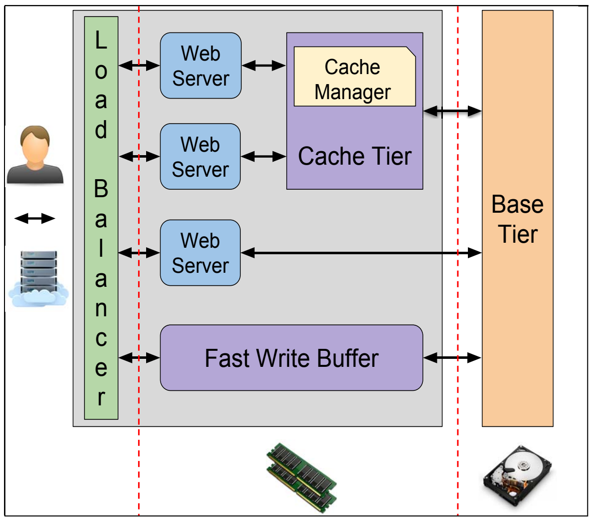

Fig. 2. Architecture of NDStore in the cloud.

spaces [32]. We utilized these indexes for these reasons in the Open Connectome Project [1]. We continue to use Morton indexes for our hierarchical data-store because their properties are applicable in the new architecture as well. Although data is stored as cuboids, we do not restrict the services to cuboid aligned regions. Users can read or write arbitrary subregions of data comprising one or more cuboids. We store a multi-resolution hierarchy for each image dataset as depicted in Figure 1. Using this hierarchy, visualization and analysis can choose the appropriate scale on which to operate.

We use a larger variant of cuboids to store the data on an object store. Per our experience, AWS S3 prefers data access to be in chunks of 16 MB. Cuboid sizes are generally smaller than this, about 256 KB. We fuse multiple cuboids into a single large cuboid, denoted *super-cuboid*, which is typically 4 times as large in each dimension and has 64 times as much data for three dimensions. Figure 4 illustrates the difference between cuboids and super-cuboids. Data is always accessed from the object store as supercuboids, ensuring optimized I/O size to S3. When loading data into the cache, we divide a super cuboid into 64 individual cuboids. Smaller sizes of cuboids are better for low latency memory access, visualization and 2-D projections. The concept of cuboids is akin to pages in virtual memory and that of supercuboids is similar to that of block sizes on file systems.

IV. Storage Architecture

The hierarchical architecture of NDStore and the interaction between the different storage tiers is presented in Figure 2. The load-balancer receives read and write requests as web-service calls from any number of sources: visualization tools, a compute cluster running HPC workflows or an individual ingesting data. Requests are redirected by the load-balancer based on the nature of the request. Read requests for immutable datasets are directed to the hierarchical storage system, fast writes to mutable datasets are directed to a memory based fast write buffer (see Section IV-C), and ingest requests are forwarded to micro-services (see Section V).

A redirecting load balancer allows for flexibility in deployment. Multiple web-servers can be deployed on demand and the caching tier can span multiple nodes. This allows the system to scale out for a surge in demand during massive HPC runs and scale back when load decreases. We allow user access to both tiers of the storage hierarchy depending on their use case. Non-recurring latency independent requests at scale for HPC workflows will be read directly from the base tier. Recurring latency sensitive requests for visualization will be read from the caching tier. NDBlaze, the fast write buffer, is deployed as a separate instance under the load-balancer for buffering random writes. All requests to data in any tier has to go through the application layer running on the web-server nodes. The application layer contains meta-data, including volumetric bounds, access-control and logic for reading and writing to unaligned regions in storage.

A. Base Storage Tier

The base tier for peta-scale neuroscience data has to be scalable, reliable and low-cost for long-term storage. I/O latency is not critical because of higher-level caches. We choose S3 as the base tier in our hierarchical storage architecture. S3 is a scalable object storage service with web interfaces. It is durable, available and secure ensuring that data is protected against disk failures. Moreover, it is a low cost option at \$0.023 per GB per month in 2018 [1]. We did consider other storage services offered by AWS, Glacier and EBS, for this tier. Glacier is a data archive and cold storage service. It has a much lower cost when compared to S3 at \$0.004 per GB per month. But, it is not a viable option because data retrievals can take up to 24 hours. EBS is a persistent block storage volume with low latency and capability to provision I/O. But storing all our data on EBS is 5 times more expensive at a cost of \$0.10 per GB per month (see Section VI-D). Moreover, there is size limit of 16 TB on each volume.

We prepend a hash to all the object keys to optimize I/O access to S3. S3, unlike a file-system, has a flat structure with no hierarchy. S3 splits it's storage partitions based on the key-space in which keys are sorted lexicographically. By prepending a random hash to the key for each supercuboid, we ensure that requests for adjacent keys are routed to different partitions. This allows I/O to be processed and delivered in parallel. Without randomization, I/O would be restricted to a partitions, each limited to 100 request per second.

1) Sparsity of Data: Many neuroscience datasets have large regions of space either empty or zero valued and we use this property to implement storage optimizations. An example of a sparse annotation dataset is shown in Figure 3, which depicts a 5000x5000 voxel section of a neuroscience annotation dataset [33]. We choose to not store any supercuboids for blank or zero data. This is helpful in reducing our storage footprint and drives down storage costs. Any absent data is dynamically

[1]The prices are for AWS *us-east-1* region as of August 2018

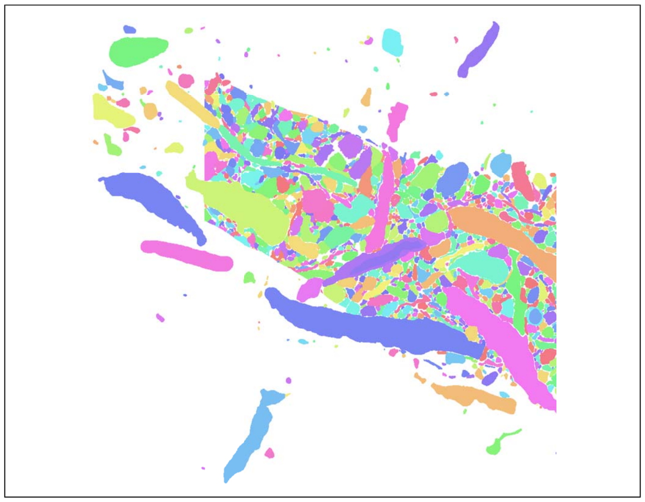
Fig. 3. An example 5000x5000 voxel section of a neuroscience annotation dataset [33] depicting sparsity.

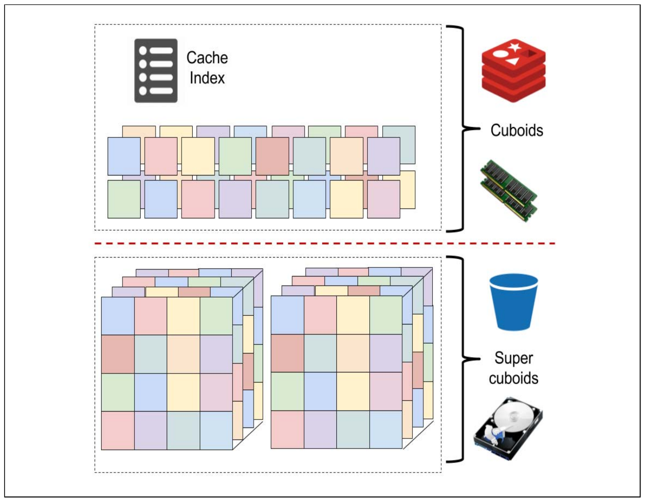

Fig. 4. Data layout across different storage tiers.

materialized on access, if the request is within the volumetric bounds of the dataset.

2) Indexing Base Contents: To deal with sparsity, we require a way to generate lists of supercuboids that contains data. This is required for data managements tasks, such as building scaling levels, data-migrations and deleting sub-sections or entire datasets. Each of these tasks requires a different supercuboid list at a different granularity level. S3 does allow a *LIST* operation over the bucket but this is cost and performance prohibitive for millions of objects and is not recommended. We could also query for each object, but every access to S3 incurs a minor cost and this can be expensive for millions of accesses to non-existing objects.

We construct and maintain an index the supercuboids in DynamoDB, which is a scalable NoSQL key-value store. It is updated at the time of supercuboid insertion into the S3 bucket. So for every supercuboid that exists in the base tier, there is a corresponding entry in DynamoDB. We include the dataset meta-data and supercuboid Morton index into the S3 object key to make them self describing. This feature allows us to generate S3 object keys without any external lookup. We use the index only for bulk data operations. For individual supercuboid reads, we query S3 directly without checking DynamoDB first. Although, there is a cost associated with a *GET* operation for S3, this is small compared to the cost of a DynamoDB access. Moreover, S3 accesses for object misses have tolerable performance.

B. Caching Tier

A caching tier for peta-scale neuroscience workloads needs to have low latency for random I/O and should be scalable to giga-bytes of data. This tier need not be low cost per GB of data because it is only holds a small fraction of the total data. Also, it can be ephemeral because we commit data to the non-volatile base tier eventually. We choose Redis [34], which is an open-source in-memory data store, as our caching tier. Redis is a good choice from the currently available commercial grade systems for multiple reasons. First, it is entirely in-memory, which offers us low latency I/O access for multiple readers and writers. Second, it has a cluster mode of operation, which allows us to scale out our caching tier and not get confined by the physical memory limitations of a single node. Third, Redis supports sorted sets and perform set operations, such as union, difference, and intersection, which we use for managing cache indexes.

Cuboids are stored in Redis as Blosc compressed [35] strings. Compressing cuboids reduces their size and, thus, their memory footprint. With reduced size per cuboid we can fit many more in the caching tier. We adopt a different approach for empty cuboids when compared to empty supercuboids. Empty cuboids are stored as empty strings in Redis. This allows us to identify that a supercuboid was fetched from the base tier but was empty. In this way, we can avoid future S3 and DynamoDB accesses while consuming very little memory.

1) Distributed Locking: We implement a distributed Readers-Writer lock on top of Redis services for data consistency in the caching tier. We write-back data to the cache and any writes to a cuboid need to be finished to memory before they can be read. The lock allows concurrent read accesses and exclusive write access. The Readers-Writer lock extends native spin lock in Redis, using Redis channels and built-in atomic operations to implement wait queues and distribution. As a Redis service, multiple processes running on different web-server nodes share the lock to realize distributed cache consistency.

2) Indexing Cache Contents: We build a cache index using sorted sets in Redis that record the contents of the cache. It is used to determine what data are missing on read requests and during cache eviction. This cache index store is different from the supercuboid index in DynamoDB that describes all supercuboids in the base tier. It consists of a project meta-data string followed by a cuboid identifier. With this format, we use the string prefix delete operation in Redis to evict multiple contiguous keys or all keys from a given data set in a single operation. Redis supports sets and sorted sets, which are an unordered collection of non-repeating strings, that provide basic set operations, such as union and intersection, in $\mathcal{O}(log(n))$ time given n elements in the set. We use the intersection operation to determine which supercuboids are missing. Similarly, we use the union operation to add supercuboid indexes when cuboids are inserted or updated. We choose sorted sets over sets because in sorted sets each element can be associated with a score, in our case this is access time. We use the score to determine the least recently accessed elements in the cache by executing a rank operation on the sorted set to select pages for eviction.

3) Cache Manager: The contents of an in-memory cache are managed based on loading and evicting supercuboids, rather than individual cuboids. This decision keeps software simple and reclaims more space with fewer evictions. If a cuboid is evicted from cache then all other cuboids within its supercuboid are also evicted. The logic here is that spatial regions tend to be accessed together (for read) and that when you evict a single cuboid you are going to have to read the entire supercuboid when you take a miss for that evicted cuboid.

Small random write accesses motivate caching cuboids at a finer granularity than the S3 supercuboid. A prevalent workload pattern in neuroscience has computer vision pipelines that reads large contiguous regions of space in an image dataset. It then detects features or structures in that data that are written out as annotations to a co-registered spatial database. There are substantial performance benefits from reading and writing smaller cuboids (256K) to memory rather than larger supercuboids (16M).

Although Redis provides a built-in LRU eviction policy, we implement a custom cache manager to enforce dependencies among all cuboids in a supercuboid. The Redis manager operates on individual keys and would evict individual cuboids. Evicting individual keys results in read-modify-write when a cuboid is dirty and other cuboids in the supercuboid have already been evicted. Our policy of evicting/loading all cuboids in a supercuboid together avoids read-modify-write to S3. Our cache manager daemon runs in the background and periodically queries Redis for it's memory usage. When the Redis memory usage exceeds an upper bound, we perform cache eviction in three steps. First, we determine the least recently used supercuboids from the cache index stored as sorted sets. Second, we lock the cache to ensure that it remains consistent and no more data is added. Third, we call a delete operation on all the cuboids associated with those supercuboids. Fourth, the cache is unlocked to resume read and write operations. We continue to loop over these four steps until the memory usage of the cache drops below our lower bound. This process competes with other I/O to ensure that a cache eviction does not starve other I/O operations.

4) Cache Operation: We manage the multiple tiers of storage for the user and present a single transparent layer for access during a read request. Figure 5 depicts the operation of the cache for a read request in 11 steps. (1) The cuboids for that region are identified using Morton indexes and are mapped to their respective supercuboid indexes. (2) We acquire a read lock on the cache for identified supercuboid indexes and (3) use the cache index store to identify the missing supercuboids. (4) The missing supercuboids are fetched from the base tier and broken into cuboids. (5) We acquire a write lock over the cache for the missing supercuboid indexes to ensure there is no inconsistency. (6) Cuboids are inserted in the caching tier and (7) the respective supercuboid indexes added to the cache index store. At this point, we update the access times on the cache indexes for those supercuboids that were requested but not fetched. This ensures that we can correctly evict least recently used supercuboids. (8) The write lock is released. (9) We read all the requested cuboids from the caching tier and (10) release the read lock. (11) The cuboids are organized into the volume requested and returned to the user.

Similarly, when a sub-region of data is written to the system, the multiple tiers of storage are transparent to the user. We identify the respective cuboids using their Morton indexes. We acquire a writer lock over the cache for the respective cuboids. These cuboids are merged with any cuboids present in caching tier and then written back to the cache. The cuboid indexes are mapped to their respective supercuboid indexes and updated in the cache index store. For cuboid indexes already present in the cache index store, the access times are updated. Finally, we release the write lock and return a success to the user.

The system also allows a direct I/O mode to read and write supercuboids to the base tier. This mode is used when ingesting data through web services and performing sequential reads. A direct write does not overwrite the object present in S3, rather it is merged with the existing object with a read-modify-write process. In this mode the cache index lookup, readers-writer lock and cache index store are all bypassed.

C. Buffer for Random Writes

We develop NDBlaze, a memory based fast write buffer, to accelerate bursts of random writes to spatial data. NDBlaze improves upon the performance of writing to the in-memory caching tier by storing written data to memory directly without de-serialization. It then asynchronously de-serializes data, merges multiple writes to the same spatial region, aligns the written data to supercuboids and writes supercuboids to the base tier. NDBlaze inherits the principle of asynchronous merging from amortized write data structures, such as the log-structured merge tree [36]. Neuroscience workloads for machine and manual annotation of data generate many small random writes, e.g. writing small patches of sparse data as seen in Figure 3. These workloads degrade I/O performance and decrease utilization of processors that have to wait for I/O. NDBlaze improves user-perceived write throughput manyfold. It is implemented with Redis and Spark.

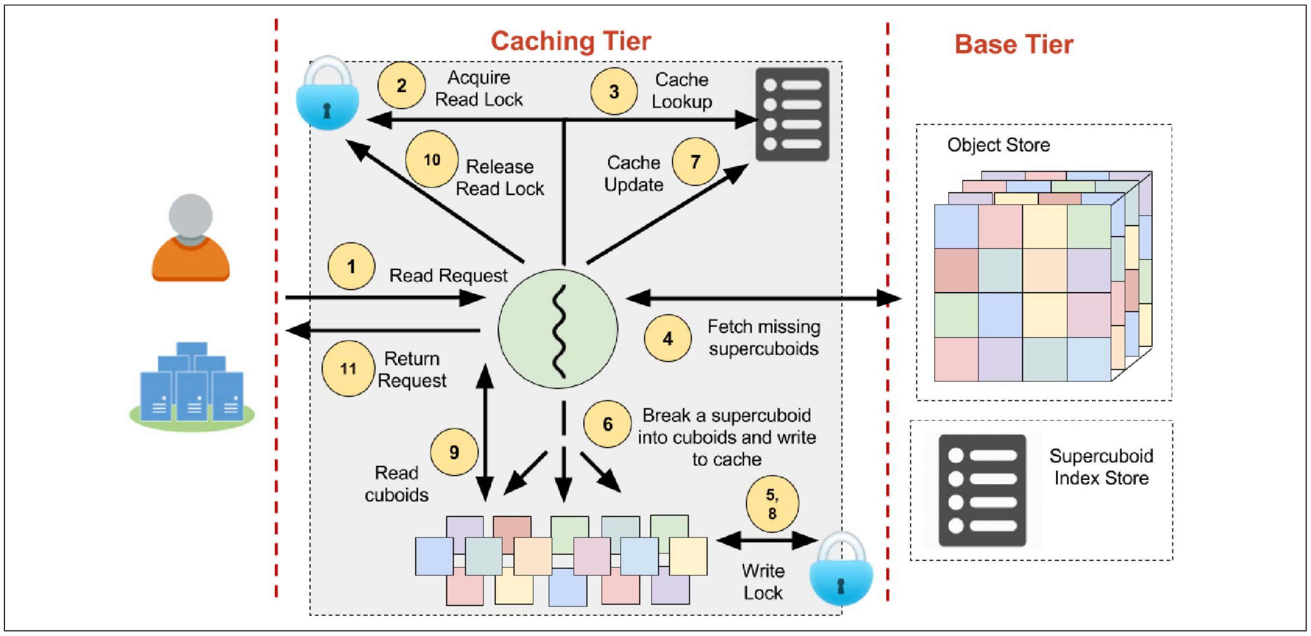

Fig. 5. Operation of the hierarchical storage model for a read request.

NDBlaze minimizes write latency and maximizes the peak throughput of write bursts. NDBlaze sits between the load balancer and the base tier. It is independent of the caching tier and utilizes its own Redis instance. This ensures that in case of a burst of random writes we do not overflow our cache and cause disruption to latency sensitive reads. We also choose not to write data from NDBlaze to the caching tier so that the asynchronous merging and write-back process reduces memory pressure. We make two key modifications for write optimization. We do not break the data blobs into cuboids before inserting them into memory. Instead we place written data into memory directly, saving time on de-serialization of data. Corresponding timestamps for each write are baked into the key of the blob to ensure correct ordering. We do build secondary index using sets in Redis to maintain a mapping between data blobs and Morton indexes. Second, there is no cache locking for writes in NDBlaze; it is unnecessary. We order the writes based on their timestamps and the secondary index determines which writes affect need to be merged into each supercuboid based Morton index. We use Spark to merge these writes. NDBlaze does allow consistent reads over the buffered data, albeit at a slower rates than the caching tier because multiple writes may need to be merged to serve a read request.

V. Microservice Processing

Several micro-services offered by AWS are used to overcome workload bottlenecks. We utilize AWS Lambda, which is an event-driven server-less computing service. It runs workflows in parallel without provisioning servers and scales applications based on the number of triggers that data requests generate. Lambda's pay-as-you-go models offers a cost benefit as well, because you pay for the number of triggered events at a milli-second granularity. We also use the AWS Simple Queue Service (SQS), reliable and scalable message queuing, to coordinate workflows across multiple servers and services in the cloud. SQS ensures that enqueued workloads are processed reliably.

A. *Data Ingest*

Modern microscopes can generate several terabytes of data in an hour [37] and storage buffers located at data collection points are not managed storage and are not large enough to store data over long periods of time. It is essential to move this data to a remote reliable data-store quickly. It is a challenge to deposit this data in the cloud, because we are limited by network and I/O speeds. We use AWS Lambda to help overcome these challenges. Figure 6 depicts a sample data ingest workflow, divided into three phases: tile collection, cuboid generation and tile cleanup.

The tile collection phase transfers microscope data (image tiles) from point of collection to the cloud. Initially, the data collection site populates the tile upload queue with a manifest of all tile names to be uploaded. Then, multiple processes at the data collection site use this task queue to select and upload tiles to an S3 bucket in parallel. The task queue ensures that after a client failure the upload process can be resumed. Every upload of tile to the tile bucket triggers a Lambda job. This Lambda job updates the received tile in an tile index DynamoDB table and removes the task from the upload queue. We maintain the tile index table to ensure that all tiles for a supercuboid are uploaded before ingest. When a Lambda job confirms that the bucket holds all tiles necessary for the

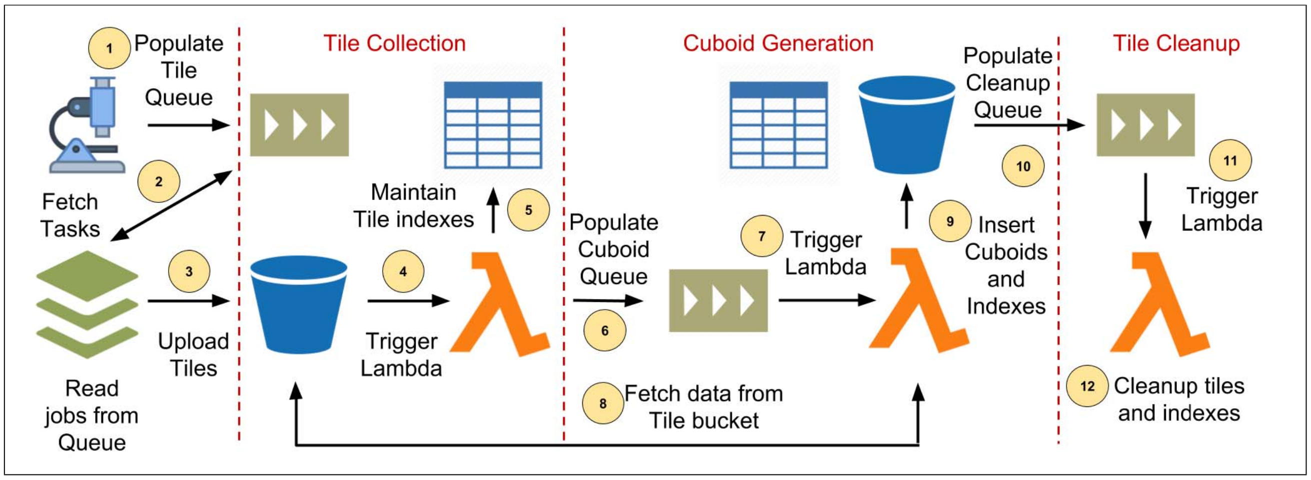

Fig. 6. Different phases of the parallel data ingest service.

supercuboid, it inserts a supercuboid generation task in the supercuboid queue and triggers another Lambda job for ingest.

The supercuboid generation phase converts the transferred tiles into supercuboids. The Lambda job reads all the relevant tiles from the tile bucket, packs them into a supercuboid, inserts the packed data into the supercuboid bucket, and updates the DynamoDB supercuboid index table. Then, this Lambda job inserts a cleanup job in the cleanup queue, triggers a cleanup Lambda job, and removes the supercuboid generation task from the cuboid queue.

The Tile Cleaning phase deletes tiles from S3 for ingested data to reclaim storage. The Lambda cleanup job removes all the tiles from the tile bucket, removes the tile indexes from the tile index table, and removes the cleanup task from the cleanup task queue.

The multiple, staggered phases of ingest ensure reliability and maximize parallelism with low resource consumption and fine-grained control. We prefer three phases to one monolithic Lambda job. Memory consumption in phases 1 and 3 is low, whereas phase 2 uses lots of memory. Different Lambda jobs settings are employed for each phase in order to fine tune memory consumption and drive down costs. We choose to dequeue tasks in each phase only when we have inserted another task in the next phase. This ensures that tasks are not lost in case of Lambda job failures. Also, each phase has different amounts or granularities of parallelism. Tile upload performs a Lambda for each tile whereas phases 2 and 3 perform a Lambda for each supercuboid. In this way, tiles can be uploaded in parallel, independently of their order in the supercuboid. This also gives a user the desired control over each phase and more Lambda jobs can be allocated for a particular phase. A user may prioritize tile collection and delay the other phases to reduce storage at the microscope. Similarly, one can delay tile-cleanup if supercuboid generation is a priority.

B. Building Scaling levels

We use DynamoDB indexing and Lambdas to generate scaling levels over the stored data. For large neuroscience data, it is customary to store data in a resolution hierarchy with each level downscaling images by a factor of 2 in all dimensions for isotropic data or just the X and Y dimensions for anisotropic data, e.g. serial section samples. This practice allows map-style visualization tools to zoom in quickly, loading the minimum amount of data for the screen resolution [6]. Every four (anisotropic) or eight (isotropic) supercuboids at any given level, create a single supercuboid at a lower level. Using the DynamoDB supercuboid index, we build a manifest of all supercuboids to be generated. For each entry in the manifest, we insert a task in an SQS propagate queue and create a Lambda job to build that cube. The SQS queue allows us to detect and rerun failed Lambdas. Lambda jobs fetch four or eight supercuboids, generate data at the lower scaling level, and insert it into S3.

VI. Experiments

The principal performance measure for NDStore is I/O throughput. We conduct throughput experiments against several different deployments; one, two and four web-server nodes with a single cache tier node running Redis. (We found that Redis provides higher throughput for a single node than in any cluster.) These experiments provide a view of the scale-up capability of NDStore. All experiments are conducted with supercuboid of 16 MB. Each thread reads a different subregion in the volume so that we do not read the same region twice and try to have all S3 reads go to disk. Cuboids and supercuboids are compressed for I/O across both tiers. Neuroscience imaging data in general has high entropy and tends to compress between 10-20% with Blosc compression used in these experiments.

A load balancer is deployed in front of our web-server nodes. We use compute optimized instances for the web-severs, general purpose instances for the Spark cluster and

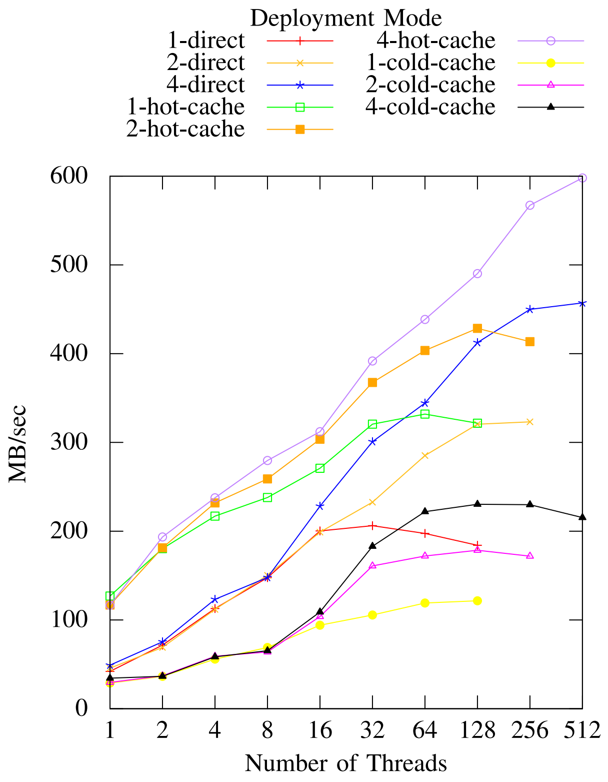

Fig. 7. Read Throughput with each thread reading 16 MB of data. The numbers 1, 2, 4 denote the number of web-servers used during benchmarks.

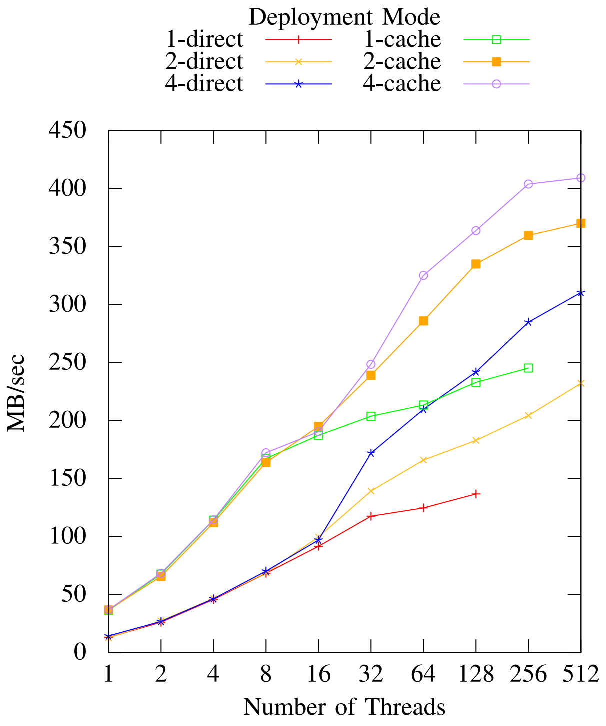

Fig. 8. Write Throughput with each threads writing 16 MB of data. The numbers 1, 2, 4 denote the number of web-servers used during benchmarks.

memory optimized instances for the Redis deployments. Each web-server node is a single c4.4xlarge instance with 16 vCPU's and 30 GB of memory. The cache tier is a single r4.4xlarge with 16 vCPU's and 100 GB of memory. NDBlaze is a single r4.4xlarge instance with 16 vCPU's and 100 GB of memory. The Spark cluster consists of a single master and four slave m4.4xlarge instances, each with 16 vCPU's and 64 GB of memory. We run a total of 16 Spark workers each with a single executor: 4 vCPU's and 16 GB of memory. We set the PySpark serializer to Kyro and the serialization buffer is allocated 1 GB. All instances have a default single root drive of 8 GB on EBS. The Lambda functions are deployed with a memory of 128 MB, minimum possible, and a maximum possible running time of 100 seconds. AWS limits concurrent Lambda job executions to a 1000 per region but these can be increased on request. We use the default limit of 1000 concurrent job executions for these experiments.

A. Read Throughput: Base Tier vs Hot Cache vs Cold Cache

NDStore achieves read throughput of 600 MB/sec for cached data from Redis and 450 MB/sec from S3. Figure 7 shows the read throughput directly from S3 compared to that from a cold and a hot Redis cache as a function of number of threads. In the case of hot cache, all of the requested data is already present in the caching tier. For cold cache experiments, the data has to be fetched from the base tier. Cold cache has the slowest performance owing to additional processing. The data are read from S3, decompressed, broken up into cuboids, re-compressed and inserted in to Redis. The cold cache peak throughput is about 225 MB/sec with 128 threads and is CPU bound. We see increases in throughput until we reach the vCPU count of the web servers, 16 per server, at which point performance flattens.

These experiments show that it is better to read directly from the base tier for non-recurring sequential reads. We achieve good I/O parallelism; the S3 object store was designed for this workload. Serving recurring reads from the cache achieves maximum performance. NDStore supports high read throughput in the cloud for a variety of workloads and knowledge of the data access pattern is important for readers to use the right web service, direct I/O to S3 or cached I/O.

B. Write Throughput: Base Tier vs Cache vs Write Buffer

Using NDBlaze, we can accept bursts of random writes that exceed the write throughput of S3. We preform two sets of experiments for write throughput. First, we test the write throughput to both tiers of our hierarchical storage system. Figure 8 show the write throughput directly to S3 compared to that from a Redis cache as a function of number of threads. A peak write throughput of about 400 MB/sec can be achieved for the caching tier with 512 threads. We achieve 300 MB/sec directly to S3. Second, we test the write throughput

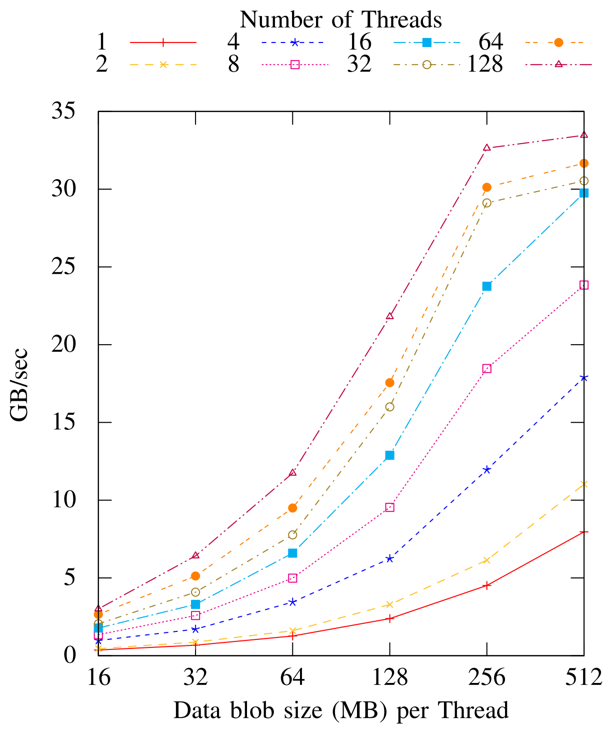

Fig. 9. Accelerated write throughput to NDBlaze using a single Redis node deployment.

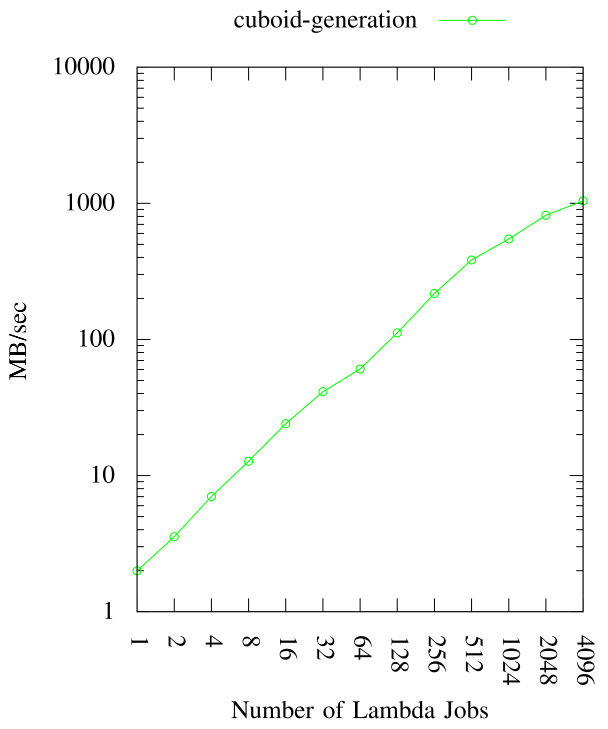

Fig. 10. Lambda memory throughput for the cuboid generation phase.

for NDBlaze, our fast write buffer. Figure 9 represents the write throughput for NDBlaze as a function of data blob size. NDBlaze achieves a peak burst throughput of about 30 GB/sec for 512 MB request with 128 threads. NDBlaze is an order of magnitude faster for write bursts when compared with the caching tier, because its write optimizations and avoidance of serialization. The time integrated performance of NDBlaze (and of the caching tier) will eventually decrease to S3 performance, because all data must eventually make it to disk.

C. Parallel ingest via Lambda

The cuboid generation phase of data ingest jobs achieves a write throughput of 1GB/sec, which exceeds the data generation rates of state-of-the-art high-throughput microscopes. Cuboid generation includes I/O to and from S3 and Lambda processing. It does not include data transfer from the microscope to S3. We choose not to measure end-to-end performance for data ingest, because network performance from neuroscience labs into the cloud tends to be quite slow and variable. Figure 10 shows the Lambda write throughput for the cuboid generation phase as a function of Lambda jobs. In this experiment, each Lambda job reads 64 512x512 PNG tiles of 256 KB each from S3. Tiles are combined to form a supercuboid of 16 MB, which is written back to S3. The write throughput scales well for about 1000 concurrent Lambda jobs and then tapers off slightly. By default, AWS limits an account to 1000 concurrent Lambda jobs, queuing additional requests. We see throughput increases beyond 1000 jobs. We see a peak throughput of 1 GB/s at 4096 concurrent Lambdas. Having outstanding Lambda requests ensures that queues are full in the presence of skew. The low cost of Lambda, \$0.000000208 per 100ms for 128 MB of memory, ensures that we ingest data at a fraction of the storage cost.

D. Storage Cost Analysis

The hierarchical storage model implemented in NDStore is 5 to 11 times cheaper per MB/sec of I/O when compared with EBS. We do a sample analysis on storing 1 TB of data based on AWS *us-east-1* region prices (as of August 2018), which is where our production system is deployed.

TABLE I
Cost Analysis of EBS vs our model for 1 TB of data. Cost for read and write are indicated by R and W respectively.

	EBS	**S3**
Storage cost per 1 TB	\$125	\$23
Operating Cost	\$1300	\$79.60
Total Cost	\$1425	\$102.60
Cost of 1 MB/sec I/O	\$4.45^{R,W}	\$0.82^{R}, \$0.41^{W}

A monthly storage cost for a 1 TB of EBS is \$125 at a price of \$0.125 per GB per month. The corresponding cost for S3 is \$23, at a price of \$0.023 per GB per month; 10

times lower than EBS. For EBS, we provision 20,000 IOPS, maximum possible per volume, so that we attain a maximum raw throughput of 320 MB/sec. This costs \$1300 per month at the price of \$0.065 per provisioned IOPS per month. The read access cost for S3 *GET*'s is \$4 a month, assuming ten million reads, at a price of \$0.0004 per 10,000 requests. The write access cost for S3 *PUT*'s is \$50 a month, assuming the same number of writes as reads, at a price of \$0.005 per 1000 requests. For the S3 approach, we incur additional costs on DynamoDB to maintain supercuboid indexes. On DynamoDB, our cost comes to about \$25.60 per month for a provisioned capacity of the 50 read and 50 write units. Our hierarchical storage system costs are about 13 times lower at \$102.60 per month as compared to EBS at \$1425 per month. Our system, assuming a single node deployment with co-located cache, can achieve a maximum read throughput of about 125 MB/sec for cold cache reads and a maximum write throughput of about 250 MB/sec to cache. Our read I/O cost is \$0.82 MB/sec per dollar and write I/O cost is \$0.41 MB/sec per dollar. The comparative read and write I/O cost for EBS is 4.45 MB/sec per dollar.

Our analysis omits compute costs on EC2. These are similar for both deployments and the amount spent on EC2 varies widely depending upon the chosen instance type. We also omit costs for NDBlaze.

VII. Conclusion and Future Work

We have developed a hierarchical storage system in the cloud, NDStore, to economically provide high I/O throughput for different neuroscience workloads. Furthermore, we utilize multiple cloud micro-services to build new services and redesign current workflows. This system is currently deployed for our project called NeuroData (http://neurodata.io) and the Open Connectome Project (http://openconnecto.me). All software described in this paper are open-source and available for collaborative development and reuse at our github repositories (NDStore: https://github.com/neurodata/ndstore, NDBlaze: https://github.com/neurodata/ndblaze, Readers-Writer Lock: https://github.com/kunallillaney/blaze-lock). Limitations on local cluster deployments and cost of a cloud deployment had been two major factors in our ability to process peta-bytes of neuroscience data. The deployment of NDStore overcomes these shortcomings and has transformed our ability to support large-scale neuroscience.

Additional engineering will allow NDStore to extend its performance even further. We have identified many possible optimizations. We plan to pre-fetch cuboids into our caching tier. This optimization will decrease cache misses and drive up read throughput. We can optimize cuboid and supercuboid aligned I/O by avoiding serialization and de-serialization of data. An aligned request will be returned to the requester. This additional overhead is currently incurred for all I/O, affecting performance. We plan to utilize Lambdas for data movement between the two tiers of NDStore. Currently, the number of processors on web-server nodes limits inter-tier data movement. Lambdas will scale-out data movement without launching new nodes and drive down compute costs. Integrating newer services, such as Lambdas with dead letter queues and step functions, will simplify workflow. Currently, we drive workflows manually through SQS and have to manage queues for completion, failure and restart. When we designed our parallel ingest process, these services were not available.

We had a positive experience building our system on the cloud and we would recommend that other open-science projects take a similar approach. Some points to keep in mind based on our experience: (1) Do not try to superimpose your current architecture on the cloud; (2) redesign the software to leverage the rich suite of services available and (3) minimize cost. A cost analysis of your workflows is essential. Cost effectiveness remains a major impediment for migration to the cloud for open-science projects. Design your system around cost and use micro-services to help keep costs low. Understand the intricacies of cloud services. Every cloud service has its own caveats and tricks. Finally, the available services keep increasing and often provide better performance or lower cost. Integrating with new services will help keep the system relevant over time.

Acknowledgements

This work was supported by grants from the National Institutes of Health (NINDS 1R01NS092474 1U01NS090449) and the National Science Foundation (IOS-1707298 ACI-1649880 OCE-1633124). The authors would like to thank additional members of the Neurodata Project that contributed to this work. We would also like to thank our collaborators Narayanan Kasthuri, Davi Bock, Forrest Collman, Wei Chung Lee, and Dan Bumbarger.

References

[1] R. Burns, K. Lillaney, D. R. Berger, L. Grosenick, K. Deisseroth, R. C. Reid, W. G. Roncal, P. Manavalan, D. D. Bock, N. Kasthuri, M. Kazhdan, S. J. Smith, D. Kleissas, E. Perlman, K. Chung, N. C. Weiler, J. Lichtman, A. S. Szalay, J. T. Vogelstein, and R. J. Vogelstein, "The Open Connectome Project Data Cluster: Scalable Analysis and Vision for High-throughput Neuroscience," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, 2013.

[2] A. S. Szalay, K. Church, C. Meneveau, A. Terzis, and S. Zeger, "MRI: The Development of Data-Scope—a multi-petabyte generic data analysis environment for science," Available at https://wiki.pha.jhu.edu/escience_wiki/images/7/7f/DataScope.pdf, 2012.

[3] W. R. G. Roncal, D. M. Kleissas, J. T. Vogelstein, P. Manavalan, K. Lillaney, M. Pekala, R. Burns, R. J. Vogelstein, C. E. Priebe, M. A. Chevillet *et al.*, "An automated images-to-graphs framework for high resolution connectomics," *Frontiers in neuroinformatics*, vol. 9, p. 20, 2015. [Online]. Available: https://www.frontiersin.org/article/10.3389/fninf.2015.00020

[4] D. M. Kleissas, W. Gray Roncal, P. Manavalan, J. T. Vogelstein, D. D. Bock, R. Burns, and R. J. Vogelstein, "Large-Scale Synapse Detection Using CAJAL3D," *Neuroinformatics*, 2013. [Online]. Available: http://www.frontiersin.org/neuroinformatics/10.3389/conf.fninf.2013.09.00037/full

[5] T. Pietzsch, S. Saalfeld, S. Preibisch, and P. Tomancak, "BigDataViewer: visualization and processing for large image data sets," *Nature Methods*, vol. 12, no. 6, pp. 481–483, 2015.

[6] "NeuroGlancer," https://github.com/google/neuroglancer. [Online]. Available: https://github.com/google/neuroglancer

[7] "Machine Intelligence from Cortical Networks (MI-CrONS) ," https://www.iarpa.gov/index.php/research-programs/microns/microns-baa, 2014. [Online]. Available: https://www.iarpa.gov/index.php/research-programs/microns/microns-baa

[8] A. Abbott *et al.*, "Solving the brain," *Nature*, vol. 499, no. 7458, pp. 272–274, 2013.

[9] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A View of Cloud Computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1721654.1721672

[10] K. R. Jackson, K. Muriki, L. Ramakrishnan, K. J. Runge, and R. C. Thomas, "Performance and Cost Analysis of the Supernova Factory on the Amazon AWS cloud," *Scientific Programming*, vol. 19, no. 2-3, pp. 107–119, 2011.

[11] 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*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 50:1–50:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413421

[12] M. C. Schatz, B. Langmead, and S. L. Salzberg, "Cloud computing and the DNA data race," *Nature Biotechnology*, vol. 28, no. 7, p. 691, 2010.

[13] J. L. Hellerstein, K. J. Kohlhoff, and D. E. Konerding, "Science in the Cloud: Accelerating Discovery in the 21st Century," *IEEE Internet Computing*, vol. 16, no. 4, pp. 64–68, July 2012.

[14] A. Thakar, A. Szalay, K. Church, and A. Terzis, "Large science databases–are cloud services ready for them?" *Scientific Programming*, vol. 19, no. 2-3, pp. 147–159, 2011.

[15] M. R. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel, "Amazon S3 for Science Grids: A Viable Solution?" in *Proceedings of the 2008 International Workshop on Data-aware Distributed Computing*, ser. DADC '08. New York, NY, USA: ACM, 2008, pp. 55–64. [Online]. Available: http://doi.acm.org/10.1145/1383519.1383526

[16] "Amazon Web Services," https://aws.amazon.com/. [Online]. Available: https://aws.amazon.com/

[17] "Microsoft Cloud," https://cloud.microsoft.com/en-us/. [Online]. Available: https://cloud.microsoft.com/en-us/

[18] "Google Cloud Platform," https://cloud.google.com/. [Online]. Available: https://cloud.google.com/

[19] "IBM Cloud," https://www.ibm.com/cloud-computing/. [Online]. Available: https://www.ibm.com/cloud-computing/

[20] C. Johnson, "IBM 3850: Mass Storage System," in *Proceedings of the May 19-22, 1975, National Computer Conference and Exposition*, ser. AFIPS '75. New York, NY, USA: ACM, 1975, pp. 509–514. [Online]. Available: http://doi.acm.org/10.1145/1499949.1500051

[21] D. W. Brubeck and L. A. Rowe, "Hierarchical Storage Management in a Distributed VOD System," *IEEE multimedia*, vol. 3, no. 3, pp. 37–47, 1996.

[22] M. N. Nelson, B. B. Welch, and J. K. Ousterhout, "Caching in the Sprite Network File System," *ACM Trans. Comput. Syst.*, vol. 6, no. 1, pp. 134–154, Feb. 1988. [Online]. Available: http://doi.acm.org/10.1145/35037.42183

[23] L.-F. Cabrera, R. Rees, S. Steiner, W. Hineman, and M. Penner, *ADSM: A Multi-Platform, Scalable, Backup and Archive Mass Storage System*. IEEE, March 1995.

[24] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, "Coda: A Highly Available File System for a Distributed Workstation Environment," *IEEE Transactions on computers*, vol. 39, no. 4, pp. 447–459, 1990.

[25] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The RAMCloud Storage System," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 7:1–7:55, Aug. 2015. [Online]. Available: http://doi.acm.org/10.1145/2806887

[26] B. Fitzpatrick, "Distributed Caching with Memcached," *Linux J.*, vol. 2004, no. 124, pp. 5–, Aug. 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=1012889.1012894

[27] P. Watson, P. Lord, F. Gibson, P. Periorellis, and G. Pitsilis, "Cloud Computing for e-Science with CARMEN," in *2nd Iberian Grid Infrastructure Conference Proceedings*. Citeseer, 2008, pp. 3–14.

[28] J. Li, M. Humphrey, C. Van Ingen, D. Agarwal, K. Jackson, and Y. Ryu, "eScience in the Cloud: A MODIS Satellite Data Reprojection and Reduction Pipeline in the Windows Azure Platform," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–10.

[29] E. Soroush, M. Balazinska, and D. Wang, "Arraystore: A Storage Manager for Complex Parallel Array Processing," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 2011, pp. 253–264.

[30] E. Perlman, R. Burns, Y. Li, and C. Meneveau, "Data Exploration of Turbulence Simulations Using a Database Cluster," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, ser. SC '07. New York, NY, USA: ACM, 2007, pp. 23:1–23:11. [Online]. Available: http://doi.acm.org/10.1145/1362622.1362654

[31] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz, "Analysis of the Clustering Properties of the Hilbert Space-Filling Curve," *IEEE Transactions on knowledge and data engineering*, vol. 13, no. 1, pp. 124–141, 2001.

[32] K. Kanov, R. Burns, G. Eyink, C. Meneveau, and A. Szalay, "Data-Intensive Spatial Filtering in Large Numerical Simulation Datasets," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. IEEE, Nov 2012, pp. 1–9.

[33] N. Kasthuri, K. J. Hayworth, D. R. Berger, R. L. Schalek, J. A. Conchello, S. Knowles-Barley, D. Lee, A. Vázquez-Reina, V. Kaynig, T. R. Jones *et al.*, "Saturated Reconstruction of a Volume of Neocortex," *Cell*, vol. 162, no. 3, pp. 648–661, 2015.

[34] S. Sanfilippo and P. Noordhuis, "Redis," http://redis.io. [Online]. Available: http://redis.io

[35] V. Haenel, "Bloscpack: a compressed lightweight serialization format for numerical data," *arXiv preprint arXiv:1404.6383*, 2014.

[36] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, Jun 1996.

[37] J. W. Lichtman, H. Pfister, and N. Shavit, "The big data challenges of connectomics," *Nature neuroscience*, vol. 17, no. 11, pp. 1448–1454, 2014.