Comparative Evaluation of Big-Data Systems on Scientific Image Analytics Workloads

Parmita Mehta, Sven Dorkenwald, Dongfang Zhao, Tomer Kaftan,
Alvin Cheung, Magdalena Balazinska, Ariel Rokem,
Andrew Connolly, Jacob Vanderplas, Yusra AlSayyad
University of Washington
{parmita, sdorkenw, dzhao, tomerk11, akcheung, mbalazin, arokem, ajc26, jakevdp, yusra}@uw.edu

ABSTRACT

Scientific discoveries are increasingly driven by analyzing large volumes of image data. Many new libraries and specialized database management systems (DBMSs) have emerged to support such tasks. It is unclear how well these systems support real-world image analysis use cases, and how performant the image analytics tasks implemented on top of such systems are. In this paper, we present the first comprehensive evaluation of large-scale image analysis systems using two real-world scientific image data processing use cases. We evaluate five representative systems (SciDB, Myria, Spark, Dask, and TensorFlow) and find that each of them has shortcomings that complicate implementation or hurt performance. Such shortcomings lead to new research opportunities in making large-scale image analysis both efficient and easy to use.

1. INTRODUCTION

With advances in data collection and storage technologies, data analysis has become widely accepted as the fourth paradigm of science [16]. In many scientific fields, an increasing portion of this data are images [21, 22]. Thus, it is crucial for big data systems[1] to provide a scalable and efficient means of storing and analyzing such data, via programming models that can be easily utilized by domain scientists (e.g., astronomers, physicists, biologists, etc).

As an example, the Large Synoptic Survey Telescope (LSST) is a large-scale international initiative to build a new telescope for surveying the visible sky [25] with plans to collect 60PB of images over 10 years. In previous astronomy surveys (e.g., the Sloan Digital Sky Survey [38]), an expert team of engineers processed the collected images on dedicated servers. The results were distilled into textual catalogs for other astronomers to analyze. In contrast, one of the goals of LSST is to broaden access to the collected images for astronomers around the globe, enabling them to run analyses on the images directly. Similarly, in neuroscience several large collections of imaging data are being compiled. For example, the UK biobank will release Magnetic Resonance Imaging (MRI) data from close to 500k human brains (more than 200TB of data) for neuroscientists to analyze [28]. Multiple other initiatives are similarly making large collections of image data available to researchers [3, 23, 42].

Such use cases emphasize the need for effective systems to support the management and analysis of image data: systems that are efficient, scale, and are easy to program without requiring deep systems expertise to deploy and tune. Surprisingly, there has been limited work from the data management research community in building systems to support large-scale image *management and analytics*. Rasdaman [34] and SciDB [36] are two well-known DBMSs that specialize in the storage and processing of multidimensional array data and are a natural choice for implementing image analytics. Besides these systems, most other work developed for storing image data targets predominantly image storage and retrieval based on keyword or similarity searches [14, 11, 12].

Hence, the key questions we ask in this paper are: How well do existing big data systems support the management and analysis requirements of real scientific image processing workloads? Is it easy to implement large-scale image analytics using these systems? How efficient are the resulting applications that are built on top of such systems? Do they require deep technical expertise to optimize?

We present the first comprehensive study of the issues mentioned above. Specifically, we picked five big data systems for parallel data processing: a domain-specific DBMS for multidimensional array data (SciDB [36]), a general-purpose cluster computing library with persistence capabilities (Spark [40]), a traditional parallel general-purpose DBMS (Myria [18, 46]), along with a general-purpose (Dask [35]) and domain-specific (TensorFlow [2]) parallel-programming library.

We selected these systems as they (1) have open-source implementations; (2) can be deployed on commodity hardware; (3) support complex analytics such as linear algebra and user defined functions; (4) provide Python APIs, which is important as the language has become immensely popular in sciences [30]; and (5) with different internal architectures such that we can evaluate the performance of different im-

[1]In this paper, we use the term "big data system" to describe any DBMS or cluster computing library that provides parallel processing capabilities on large amounts of data.

Proceedings of the VLDB Endowment, Vol. 10, No. 11

plementation paradigms. We discuss in more detail reasons for choosing these systems in Section 2.

To evaluate these systems, we implement two representative end-to-end image analytics pipelines from astronomy and neuroscience. Each pipeline has a reference implementation in Python provided by domain scientists. We then attempt to re-implement them using the five big data systems described above and deploy the resulting implementation on commodity hardware in the Amazon Web Services cloud [5], to simulate the typical hardware and software setup in scientists' labs. We evaluate the resulting implementations with the following goals in mind:

- Investigate if the given system can be used to implement the pipelines and, if so, how easy is it to do so (Section 4).
- Measure the execution time of the resulting pipelines in a cluster deployment (Section 5.1 and Section 5.2).
- Evaluate the system's ability to scale, both with the number of machines available in the cluster, and the size of the input data to process (Section 5.1).
- Assess tuning required by each system to correctly and efficiently execute each pipeline (Section 5.3).

Our study shows that, despite the difference in domains, both real-world use cases have important similarities: input data is in the form of multidimensional arrays encoded in domain-specific file formats (FITS [15], NIfTI [29], etc.); data processing involves slicing along different dimensions, aggregations, stencil (a.k.a.multidimensional window) operations, spatial joins and complex transformations expressed in Python.

Overall, we find that all systems have important limitations. While performance and scalability results are promising, there is much room for improvement in usability and efficiently supporting image analytics at scale.

2. EVALUATED SYSTEMS

In this section we briefly describe the five evaluated systems and their design choices pertinent to image analytics. The source code for all of these systems is publicly available.

Dask [1] (v0.13.0) is a general-purpose parallel computing library implemented entirely in Python. We select Dask because the use cases we consider are written in Python. Dask represents parallel computation with task graphs. Dask supports parallel collections such as `Dask.array` and `Dask.dataframe`. Operations on these collections create a task graph implicitly. Custom (or existing) code can be parallelized via `Dask.delayed` statements, which delay function evaluations and insert them into a task graph. Individual task(s) can be submitted to the Dask scheduler directly. Submitted tasks return `Futures`. Further tasks can be submitted on `Futures`, sending computation to the worker where the `Future` is located. The Dask library includes a scheduler that dispatches tasks to worker processes across the cluster. Processes execute these tasks asyncronously. Dask's scheduler determines where to execute the delayed computation, and serializes the required functions and data to the chosen machine before starting its execution. Dask uses a cost-model for scheduling and work stealing among workers. The cost-model considers worker load and user specified restrictions (e.g., if a task requires a worker with GPU) and data dependencies of the task. Computed results remain on the worker where the computation took place and are brought back to the local process by calling `result()`.

Myria [18, 46] is a relational, shared-nothing DBMS developed at the University of Washington. Myria uses PostgreSQL [33] as its node-local storage subsystem and includes its own query execution layer on top of it. Users write queries in MyriaL, an imperative-declarative hybrid language, with SQL-like declarative constructs and imperative statements such as loops. Myria query plans are represented as graphs of operators and may include cycles. During execution, operators pipeline data without materializing it to disk. Myria supports Python user-defined functions and a BLOB data type. The BLOB data type allows users to write queries that directly manipulate objects like NumPy arrays. We select Myria as a representative shared nothing parallel DBMS and also a system that we built. Our goal is to understand how it compares to other systems on image analysis workloads and what it requires to effectively support such tasks.

Spark [47] (v1.6.2) supports a dataflow programming paradigm. It is up to 100× faster than Hadoop for in-memory workloads and up to 10× faster for workloads that persist to disk [40]. We select Spark for its widespread adoption, its support for a large variety of use cases, the availability of a Python interface, and to evaluate the suitability of the MapReduce programming paradigm for large-scale image analytics. Spark's primary abstraction is a distributed, fault-tolerant collection of elements called Resilient Distributed Datasets (RDDs) [47], which are processed in parallel. RDDs can be created from files in HDFS or by transforming existing RDDs. RDDs are partitioned and Spark executes separate tasks for different RDD partitions. RDDs support two types of operations: transformations and actions. Transformations create a new dataset from an exisiting one, e.g., `map` is a transformation that passes each element through a function and returns a new RDD representing the results. Actions return a value after running a computation: e.g., `reduce` is an action that aggregates all the elements of the RDD using a function and returns the final result. All transformations are lazy and are executed only when an action is called. Besides `map` and `reduce`, Spark's API supports relational algebra operations as well, such as distinct, union, join, grouping, etc.

SciDB [10] (v15.12) is a shared-nothing DBMS for storing and processing multidimensional arrays. SciDB is designed specifically to support image analytics tasks such as those we evaluate in this paper, and is an obvious system to include in the evaluation. In SciDB, data is stored as arrays divided into chunks distributed across nodes in a cluster. Users then query the stored data using the Array Query Language (AQL) or Array Functional Language (AFL) through the provided Python interface. SciDB supports user-defined functions in C++ and, recently, Python (with the latter executed in a separate Python process). In SciDB, query plans are represented as an operator tree, where operators, including user-defined ones, process data iteratively one chunk at a time.

TensorFlow [2] (v0.11) is a library from Google for numerical computation using dataflow graphs. Although TensorFlow is typically used for machine learning, it supports a wide range of functionality to express operations over N-

dimensional tensors. Such operations are organized into dataflow graphs, where nodes represent computations, and edges are the flow of data expressed using tensors. TensorFlow optimizes these computation graphs and can execute them locally, in a cluster, on GPUs, or on mobile devices. We select TensorFlow to evaluate its suitability for large-scale image analytics given its general popularity.

3. IMAGE ANALYTICS USE CASES

Scientific image data is available in many modalities: X-ray, MRI, ultrasound, telescope, microscope, satellite, etc. We select two scientific image analytics use cases from neuroscience and astronomy, with real-world utility in two different domains. The choice of these use cases is influenced by access to domain experts in these fields, availability of large datasets, and interest from the domain experts in scaling their use cases to larger datasets with big data systems. Reference implementations for both use cases are provided by domain experts who are also coauthors on this paper, and were written for their prior work. Both reference implementations are in Python and utilize wrapped libraries written in C/C++. We present the details of the use cases in this section. Both use cases are subdivided into steps. We use Step Ⓧ$_N$ and Step Ⓧ$_A$ to denote steps in the neuroscience and astronomy use cases respectively.

3.1 Neuroscience

Many sub-fields of neuroscience use image data to make inferences about the brain [24]. The use case we focus on analyzes Magnetic Resonance Imaging (MRI) data in human brains. Specifically, we focus on measurements of diffusion MRI (dMRI), an imaging technology that is sensitive to water diffusion in different directions within a human brain. These measurements are used to estimate large-scale brain connectivity, and to relate the properties of brain connections to brain health and cognitive functions [45].

Data: The input data comes from the Human Connectome Project [44]. We use data from the S900 release, which includes dMRI data from over 900 healthy adult subjects collected between 2012 and 2015. The dataset contains dMRI measurements obtained at a spatial resolution of 1.25×1.25×1.25 mm^3. Measurements were repeated 288 times in each subject, with different diffusion sensitization in each measurement. The data from each measurement, called a *volume*, is stored in a 3D (145×145×174) array of floating point numbers, with one value per three-dimensional pixel (a.k.a.voxel). Image data is stored in the standard NIfTI-1 file format used for neuroimaging data. Additional metadata about the measurement used during analysis is stored in text files that accompany each measurement. We use data from 25 subjects for this use case. Each subject's data is ∼1.4GB in compressed form, which expands to ∼4.2GB when uncompressed. The total amount of data from 25 subjects is thus approximately **105GB**.

Processing Pipeline: The benchmark contains three steps from a typical dMRI image analysis pipeline for each subject, as shown in Figure 1.

1. **Segmentation**: Step ①$_N$ constructs a 3D mask that segments each image volume into two parts: the brain and the background. As the brain comprises around two-thirds of the image volume, using the mask to filter out the background will speed up subsequent steps.

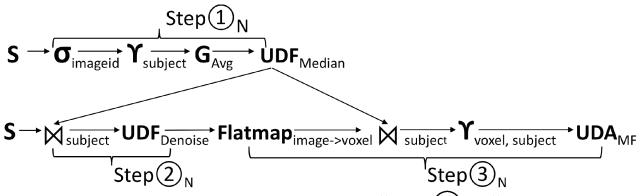


Figure 1: Neuroscience use case: Step ①$_N$ Segmentation, Step ②$_N$ Denoising, and Step ③$_N$ Model fitting.

   Segmentation proceeds in three sub-steps. First, we select a subset of the volumes where no diffusion sensitization has been applied. These images are used for segmentation as they have higher signal-to-noise ratio. Next, we compute a mean image from the selected volumes by averaging the value of each voxel. Finally, we apply the Otsu segmentation algorithm [31] to the mean volume to create a mask volume per subject.

2. **Denoising**: Denoising is needed to improve image quality and accuracy of the analysis results. This step (Step ②$_N$) can be performed on each volume independently. Denoising operates on a 3D sliding window of voxels using the non-local means algorithm [13] and uses the mask from Step ①$_N$ to denoise only parts of the image volume containing the brain.

3. **Model fitting**: Finally, Step ③$_N$ computes a physical model of diffusion. We use the diffusion tensor model (DTM) to summarize the directional diffusion profile within a voxel as a 3D Gaussian distribution [6]. Fitting the DTM is done independently for each voxel and can be parallelized. This step consists of a flatmap operation that takes a volume as input and outputs multiple voxel blocks. All 288 values for each voxel block are grouped together before fitting the DTM for each voxel. Given the 288 values in for each voxel, fitting the model requires estimating a 3×3 variance/covariance matrix. The model parameters are summarized as a scalar for each voxel called Fractional Anistropy (FA) that quantifies diffusivity differences across different directions.

The reference implementation is written in Python and Cython using Dipy [17].

3.2 Astronomy

As discussed in Section 1, astronomy surveys are generating an increasing amount of image data. Our second use case is an abridged version of the LSST image processing pipeline [26], which includes analysis routines used by astronomers.

Data: We use data from the High Cadence Transient Survey [20] for this use case, as data from the LSST survey is not yet available. A telescope scans the sky through repeated *visits* to individual, possibly overlapping, locations. We use up to 24 visits that cover the same area of the sky in this evaluation. Each visit is divided into 60 *images*, with each consisting of an 80MB 2D image (4000×4072 pixels) with associated metadata. The total amount of data from all 24 visits is approximately **115GB**. The images are encoded using the FITS file format [15] with a header and data block. The data block has three 2D arrays, with each element containing flux, variance, and mask for every pixel.

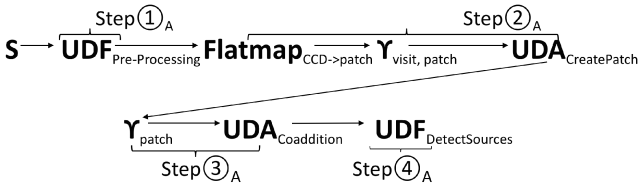


Figure 2: Astronomy use case: Step ①A Pre-Processing, Step ②A Patch Creation, Step ③A Co-addition, and Step ④A Source Detection.

Processing Pipeline Our benchmark contains four steps from the LSST processing pipeline as shown in Figure 2:

1. **Pre-Processing**: Step ①A pre-processes each image to remove background noise and artifacts caused by imaging instruments. These operations are implemented as convolutions with different kernels. This step can be parallelized across images.

2. **Patch Creation**: Step ②A re-grids the pre-processed images of the sky into regions called *patches*. Each image can be part of 1 to 6 patches requiring a flatmap operation. As pixels from multiple images can contribute to a single patch, this step groups the images associated with each patch and creates a new image object for each patch in each visit.

3. **Co-addition**: Step ③A groups the images associated with the same patch across different visits and stacks them by summing up the pixel (or flux) values. This is called co-addition and is performed to improve the signal to noise ratio of each image. Before summing up the pixel values, this step performs iterative outlier removal by computing the mean flux value for each pixel and setting any pixel that is three standard deviations away from the mean to null. Our reference implementation performs two such cleaning iterations.

4. **Source Detection**: Finally, Step ④A detects sources visible in each co-added image generated from Step ③A by estimating the background and detecting all pixel clusters with flux values above a given threshold.

The reference implementation is written in Python, and depends on several libraries implemented in C++, utilizing the LSST stack [25]. While the LSST stack can run on multiple nodes, the reference is a single node implementation.

4. QUALITATIVE EVALUATION

We evaluate the five big data systems along two dimensions. In this section, we discuss each system's ease of use and overall implementation complexity. We discuss performance and required physical tunings in Section 5. A subset of the computer science coauthors implemented each use case on each system based on the reference implementations provided by the domain expert coauthors. The team had previous experience with Myria, Spark, and SciDB but not Dask or TensorFlow.

4.1 Dask

Implementation: As described in Section 2, users specify their Dask computation using task graphs. However, unlike other big data systems with task graphs, users do not need to use a specific data abstraction (e.g., RDDs, relations, tensors, etc.). They can construct graphs directly with Python data structures. As an illustration, Figure 3 shows a code fragment from Step ①N of the neuroscience use case. We first construct a compute graph that downloads and filters each subject's data on Line 2. Note the use of `delayed` to construct a compute graph by postponing computation, and specifying that `downloadAndFilter` is to be called on each subject separately. At this point, only the compute graph is built but it has not been submitted, i.e., data has not been downloaded or filtered. Next, on Line 5 we request that Dask evaluate the graph via the call to `result` to compute the number of volumes in each subject's dataset. Calling `result` submits the graph and forces evaluation. We constructed the graph such that `downloadAndFilter` is called on individual subjects. Dask will parallelize the computation across the worker machines and will adjust each machine's load dynamically. We then build a new graph to compute the average image of the volumes by calling `mean`. We intend this computation to be parallelized across blocks of voxels, as indicated by the iterator construct on Line 9. Next, the individual averaged volumes are reassembled on Line 10, and calling `median_otsu` on Line 11 computes the mask. The rest of the neuroscience use case follows the same programming paradigm, and we implemented the astronomy use case similarly.

```
1  for id in subjectIds:
2    data[id].vols = delayed(downloadAndFilter)(id)
3
4  for id in subjectIds: # barrier
5    data[id].numVols = len(data[id].vols.result())
6
7  for id in subjectIds:
8    means = [delayed(mean)(block) for block in
9             partitionVoxels(data[id].vols)]
10   means = delayed(reassemble)(means)
11   mask = delayed(median_otsu)(means)
```

Figure 3: Dask code for Step ①N.

Qualitative Assessment: We find that Dask has the simplest installation and deployment. As Dask supports external Python libraries we reuse most of the reference implementation. Our Dask implementation has approximately 120 lines of code (LoC) for the neuroscience use case and 260 LoC for the astronomy one. When implementing compute graphs, however, a developer has to reason about when to insert barriers to evaluate the constructed graphs. Additionally, the developer must manually specify how data should be partitioned across different machines for each of the stages to facilitate parallelism (e.g., by image volume or blocks of voxels, as specified on Line 9 in Figure 3). While Dask's Python interface might be familiar to domain scientists, the explicit construction of compute graphs to parallelize computation is non-trivial. Additionally, as the Dask API supports multiple ways to parallelize code, choosing among them can impact the correctness and performance of the resulting implementation. For instance, having to choose between `futures` and `delayed` constructs to create a task graph implicitly and explicitly make Dask more flexibile but harder to use. Dask is also hard to debug due to its atypical behavior and instability. For instance, rather than failing a job after a large enough number of workers die, Dask respawns the killed processes but queues tasks executed on the killed worker processes to a `no-worker` queue.

```
conn = MyriaConnection(url="...")
MyriaPythonFunction(Denoise, outType).register()
query = MyriaQuery.submit("""
  T1 = SCAN(Images);
  T2 = SCAN(Mask);
  Joined = [SELECT T1.subjId, T1.imgId, T1.img, T2.mask
            FROM T1, T2
            WHERE T1.subjId = T2.subjId];
  Denoised = [FROM Joined EMIT
              Denoise(T1.img, T1.mask) as
              img, T1.subjId, T1.imgId]; """ )
```

Figure 4: Myria code for Step ②$_N$.

The `no-worker` state implies that tasks are ready to be computed, but no appropriate worker exists. As the running processes wait for the results from the requeued tasks this causes a deadlock. We also experienced several stability issues, some of which prevented us from running the astronomy use case initially, but were fixed in a later Dask version. We still had to stop and rerun the pipelines occasionally as they would freeze for unknown reasons.

4.2 Myria

Implementation: We use MyriaL to implement both pipelines, with calls to Python UDF/UDAs for all core image processing operations. In the neuroscience use case, we ingest the input data into the `Images` relation, with each tuple consisting of subject ID, image ID, and image volume (a serialized NumPy array stored as a BLOB) attributes. We execute a query to compute the mask, which we broadcast across the cluster. The broadcast table is 0.3% the size of the `Images` relation. A second query then computes the rest starting from a broadcast join between the data and the mask. Figure 4 shows the code for denoising image volumes. Line 1 to Line 2 connect to Myria and register the denoise UDF. Line 3 then executes the query to join the `Images` relation with the `Mask` relation and denoise each image volume. We implement the astronomy use case similarly using MyriaL.

Qualitative Assessment: Our MyriaL implementation leverages the reference Python code, facilitating the implementation of both use cases. We implement the neuroscience use case in 75 LoC and the astronomy one in 250. MyriaL's combination of declarative query constructs and imperative statements makes it easy to specify the analysis pipelines. However, for someone not familiar with SQL-like syntax this might be a challenge. Overall, implementing the use cases in Myria was easy but making the implementation efficient was non-trivial, as we discuss in Section 5.3.

4.3 Spark

Implementation: We use Spark's Python API to implement both use cases. Our implementation transforms the data into Spark's pair-RDDs, which are parallel collections of key-value pair records. The key for each RDD is the identifier for an image fragment, and the value is a NumPy array with the image data. Our implementation then uses the predefined Spark operations (`map`, `flatmap`, and `groupby`) to split and regroup image data following the plan from Figure 1. To avoid joins, we make the mask, which is a single image per subject (18MB per subject, 0.3% of the image RDD) a broadcast variable available on all workers. We use the Python functions from the reference implementation to perform the actual computations on the values, passing them as anonymous functions to Spark's API. Figure 5

```
modelsRDD = imgRDD
  .map(lambda x:denoise(x,mask))
  .flatMap(lambda x: repart(x, mask))
  .groupBy(lambda x: (x[0][0],x[0][1]))
  .map(regroup)
  .map(fitmodel)
```

Figure 5: Spark code showing Step ②$_N$ and Step ③$_N$.

```
from scidbpy import connect
sdb = connect('...')
data_sdb = sdb.from_array(data)
data_filtered = # Filter
  data_sdb.compress(sdb.from_array(gtab.b0s_mask), axis=3)
mean_b0_sdb = data_filtered.mean(index=3) # Mean
```

Figure 6: SciDB implementation of Step ①$_N$.

shows an abridged version of the code for the neuroscience use case. Line 2 denoises each image volume. Line 3 calls `repart` to partition each image volume into voxel groups, which are then grouped (line 4) and aggregated (line 5). Line 6 then fits a DTM for each voxel group. We implement the astronomy use case similarly.

Qualitative Assessment: Spark can execute user-provided Python code as UDFs and its support for arbitrary python objects as keys made the implementation straightforward, with 114 and 238 LoC for the neuroscience and astronomy use cases respectively. The functional programming style used by Spark is succinct, but can pose a challenge if the developer is unfamiliar with functional programming. Spark's large community of developers and users, and its extensive documentation are a considerable advantage. Spark supports caching data in memory but does not store intermediate results by default. Unless data is specifically cached, a branch in the computation graph may result in re-computation of intermediate data. In our use cases, opportunities for data reuse are limited. Nevertheless, we found that caching the input data for the neuroscience use case yielded a consistent 7-8% runtime improvement across input data sizes. Caching did not improve the runtime of the astronomy use case. As with Myria, the initial implementation was easy in Spark, but performance tuning was not always straightforward as we describe in Section 5.3.

4.4 SciDB

Implementation: SciDB is designed for array analytics to be implemented in AQL/AFL, or optionally using SciDB's Python API. This is illustrated in Figure 6. When we began this project, SciDB lacked several functions necessary for the use cases. For example, high-dimensional convolutions are not implemented in AFQ/AFL, rendering the reimplementation of some steps impossible. SciDB recently added a `Stream()` interface, similar to Hadoop Streaming. This interface enables the execution of Python (or other language) UDFs, where each UDF call processes one array chunk. Using this feature, we implemented both use cases in their entirety. The SciDB implementation of the neuroscience and astronomy use cases required 155 LoC and 715 LoC of Python respectively. In addition, 200 LoC of AFL were written to rearrange chunks, and another 150 LoC in bash to set up the environment required by the application.

Qualitative Assessment: The recent `stream()` interface makes it possible to execute legacy Python code, but it requires that data be passed to the Python UDF and returned from that UDF as a string in TSV format (data ingest requires a CSV format). This means that we need to convert between TSV and scientific image formats. While this is relatively straightforward for the neuroscience use case, FITS files used in astronomy have multiple arrays per image and are more challenging to transform between SciDB arrays and TSVs for UDFs. This transformation and subsequent parsing also causes performance issues as we discuss in Section 5.3. Setting up SciDB is also difficult as there is no support for standard cloud deployment tools. The integration with the LSST software stack for the astronomy use case is particularly challenging. Specifically, LSST's software stack requires dozens of dependent packages to be installed, along with setting up more than 100 environment variables within child processes that execute the UDFs. Another limitation of the `stream()` interface lies in its input and output ports: All the input can be read only through the standard input (i.e., stdin) and all the output can only be written to standard output (i.e., stdout). Therefore, the application crashes if it calls UDFs that have their own stdout messages. Overall, SciDB presents significant barriers in implementation and setup for a domain scientist.

4.5 TensorFlow

Implementation: TensorFlow's API operates on its own data structures (representations of multi-dimensional arrays, or tensors) and does not allow any of the standard external libraries to be used, such as NumPy. Hence, we had to completely rewrite the use cases. Given their complexity, we only implemented the neuroscience use case. Additionally, we implemented a somewhat simplified version of the final mask generation operation in Step ①$_N$. We rewrote Step ②$_N$ using convolutions, but without filtering with the mask as TensorFlow does not support element-wise data assignment. TensorFlow's support for distributed computation is currently limited. The developer must manually map computation and data to each worker as TensorFlow does not provide automatic static or dynamic work assignment. As the serialized compute graph cannot be larger than 2GB, we implemented the use case in steps, building a new compute graph for each step of the use case (as shown in Figure 1). We distribute the data for each step to the workers and execute it. We add a global barrier to wait for all workers to return their results before proceeding. The master node converts between NumPy arrays and tensors as needed. Figure 7 shows the code for mean computation in Step ①$_N$. The first loop assigns the input data with shape `sh` (Line 8) and the associated code (Line 9) to each worker. Then, we process the data in batches of size equal to the number of available workers on Line 19.

Qualitative Assessment: TensorFlow requires a complete rewrite of the use cases, which we find to be both difficult and time-consuming. TensorFlow also requires that users manually specify data distribution across machines and it automates only a small part of the cluster initialization procedure with Bazel [7]. The 2GB limit on graph size further complicates the implementation of use cases as we described above. In its current form, unless there is a specific reason, e.g., a specific algorithm only available in the TensorFlow library, or need to utilize GPUs or Android devices, the amount of effort required to re-write the computation in Tensorflow is high enough to render it not worthwhile. However, TensorFlow is under active development so this might change in future versions.

```
1  # steps contains the predefined mapping
2  # from data to workernodes
3  pl_inputs = []
4  work = []
5  # The first for loop defines the graph
6  for i_worker in range(len(steps[0])):
7    with tf.device(steps[0][i_worker]):
8      pl_inputs.append(tf.placeholder(shape=sh))
9      work.append(tf.reduce_mean(pl_inputs[-1]))
10 mean_data = []
11 # Iterate over the predefined steps
12 for i in range(len(steps)):
13   this_zs = zs[i*len(steps[0]):
14                i*len(steps[0]) + len(steps[i])]
15   # Define the input to be fed into the graph
16   zip_d = zip(pl_inputs[:len(steps[i])],
17               [part_arrs[z] for z in this_zs])
18   # Executes the actual computation - incl. barrier
19   mean_data += run(work[:len(steps[i])], zip_d)
```

Figure 7: TensorFlow code fragment showing compute graph construction and execution.

5. QUANTITATIVE EVALUATION

We evaluate the performance of the implemented use cases and the system tunings necessary for successful and efficient execution. All experiments are performed on the AWS cloud using the r3.2xlarge instance type, with 8 vCPU,[2] 61GB of memory, and 160GB SSD storage.

5.1 End-to-End Performance

We first evaluate the performance of running the two use cases end-to-end. For each use case, we start the execution with data stored in Amazon S3. We execute all the steps and leave the final output in worker memories. We ask two questions: How does the performance compare across the systems? How well do the systems scale as we increase the size of the input data or the cluster? To answer these questions, we first fix the cluster size at 16 nodes and vary the input data size. For the neuroscience use case we vary the number of subjects from 1 to 25. For the astronomy use case, we vary the number of visits from 2 to 24. The largest input data sizes are then 105GB and 115GB, respectively as shown in Figure 8a and Figure 8b. The tables also show the sizes of the largest intermediate relations for the two use cases, which are 210GB and 288GB, respectively. In the second experiment, we use the largest input data size for each use case and vary the cluster size from 16 to 64 nodes to measure speedup.

Figure 8c and Figure 8f show the results as we vary the input data size. We implement the neuroscience use case in all five systems and the astronomy use case in all but TensorFlow.

For the neuroscience use case (Figure 8c), while Dask, Myria and Spark achieve comparable performance, SciDB and TensorFlow are much slower. It is not surprising that Dask, Spark, and Myria have similar runtimes as all three execute the same Python code from the reference implementation but wrapped in UDFs (or directly in the case of Dask). Interestingly, the fact that Spark and Myria incur

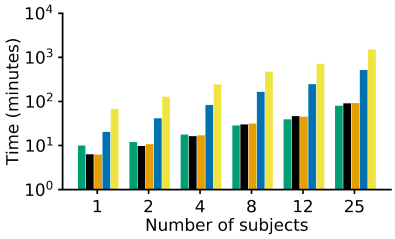
[2]with Intel Xeon E5-2670 v2 (Ivy Bridge) processors.

Subjects	1	2	4	8	12	25
Input	4.1	8.4	16.8	33.6	50.4	105
Largest inter.	8.4	16.8	33.6	67.2	100.8	210

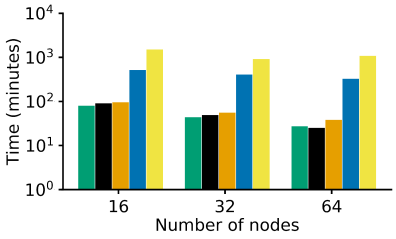
(a) Neuroscience data sizes (GB)

Visits	2	4	8	12	24
Input	9.6	19.2	38.4	57.6	115.2
Largest inter.	24	48	96	144	288

(b) Astronomy data sizes (GB)



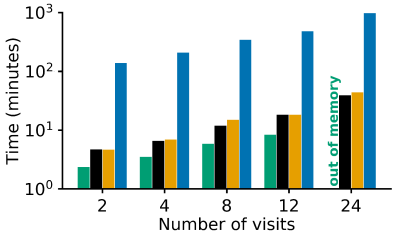
(c) Neuroscience: End-to-end runtime with varying data size



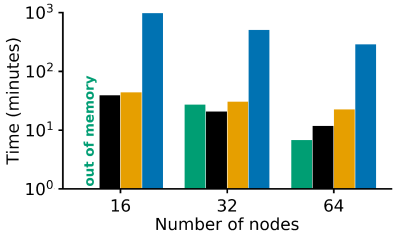
(d) Neuroscience: End-to-end runtime with varying cluster size

Subjects	1	2	4	8	12	25
SciDB	1	1.01	1.01	1.01	1.00	1.00
Spark	1	0.86	0.68	0.63	0.60	0.59
Myria	1	0.77	0.64	0.60	0.61	0.58
Dask	1	0.60	0.45	0.36	0.33	0.32
TensorFlow	1	0.95	0.90	0.88	0.89	0.89

(e) Neuroscience: Normalized runtime per subject



(f) Astronomy: End-to-end runtime with varying data size



(g) Astronomy: End-to-end runtime with varying cluster size

Visits	2	4	8	12	24
SciDB	1	0.74	0.62	0.58	0.58
Spark	1	0.74	0.80	0.65	0.78
Myria	1	0.70	0.64	0.65	0.69
Dask	1	0.75	0.62	0.59	–

(h) Astronomy: Normalized runtime per visit

Figure 8: Overall performance: results for end-to-end experiments for Neuroscience and Astronomy use cases.

the extra overhead of passing the data to and from external Python processes for each UDF invocation does not visibly affect performance. Dask is a little slower in the case of a single subject because the data is downloaded on a single node first before being spread across the cluster through work stealing as discussed in Section 5.2. SciDB is slower primarily due to data reformatting at the input and output of each UDF and also at the beginning of the pipeline: (1) SciDB's data ingest requires the source data to be in CSV format, and in both use cases, we needed to convert the original formats to CSV before executing the pipeline. We discuss data ingest in more detail when we present Figure 9. (2) When passing data from SciDB to the UDFs through the `stream()` interface, the data gets flattened into a long 1-dimensional array and we need to reformat it into the multidimensional NumPy array required by the function. (3) The results returned from each UDF are arrays of strings, one per parallel UDF invocation and need to be parsed and merged to form voxel batches from image volumes. Additionally, we process one subject at a time in SciDB instead of merging data from all subjects into a single array as combining multiple subjects incurs more overhead. For example, when we combine two subjects into one array, the execution time doubled (not shown in the figure), relative to processing one subject at a time. TensorFlow is the slowest to execute the neuroscience end-to-end pipeline. We attribute this to several architectural restrictions. (1) All data needs to be downloaded and parsed on the master node and then dispatched to the workers, as opposed to being downloaded and processed directly by the workers as in the other systems. (2) Due to TensorFlow's limitation on the computation graph size (2GB), we construct multiple graphs for each step and put the results together on the master node. This adds significant overhead at each step and prevents TensorFlow from pipelining the computation. We further analyze per-step performance in the next section.

For the astronomy use case (Figure 8f), SciDB is an order of magnitude slower than the other engines. In Step ③$_A$, we need to add patches from all visits. This means that we have to process all visits as a single array, leading to large overheads due to parsing and combining the output of the UDF in Step ②$_A$. We also implemented Step ③$_A$ in AQL in SciDB, which performed as fast as Spark, Myria and Dask, i.e., $\sim 10\times$ faster than the `stream()` implementation. As we were unable to implement the rest of the use case in AQL/AFL, we use the `stream()` timing in Figure 8.

In the astronomy use case, Dask does better with the smaller datasets but is unable to execute the largest data set to completion as it runs out of memory on 16 nodes. We discuss out of memory issues in Section 5.3. In terms of runtime, data for the astronomy use case is packaged into individual files for each image rather than a single compressed file per subject as in the neuroscience use case, which results in finer-grained and more even data distribution.

Figure 8e and Figure 8h show the runtimes per subject and visit, respectively, i.e., the ratios of each pipeline runtime to that obtained for one subject or visit. As the data size increases, these ratios drop for all systems with the exception of SciDB in the neuroscience use case. The dropping ratios indicate that the systems become more efficient as they amortize start-up costs. Dask's efficiency increase is most pronounced, indicating that it had the largest start-up overhead. In the neuroscience use case, SciDB shows a constant ratio. This is due to the neuroscience use case implementation in SciDB where the cluster processes a single subject at a time.

Figure 8d and Figure 8g show the runtimes for all systems as we increase the cluster size and process the largest (>100GB) datasets. All systems show near linear speedup for both use cases. Myria achieves almost perfect linear speedup for both cases (5395s, 2863s, and 1406s for 16, 32 and 64 nodes, respectively). Linear speedup can imply poor

single instance performance. To ensure that this is not the case, we tune single instance performance for Myria by increasing the number of workers until each node has high resource utilization (average CPU utilization > 90%). Dask has better performance when data completely fits in memory, as in the neuroscience use case. For the astronomy use case, Dask runs out of memory on 16 nodes. For 32 nodes, the number of workers had to be reduced to one per node to allow Dask to finish processing. This results in lowered parallelism and higher runtime. In the 64 node experiment there was enough memory and every step could be pipelined for both Myria and Dask, resulting in faster runtimes. Spark divides each task into stages and does not start shuffling data until the previous map stage is complete. As the tuples are large in size, (∼14MB for neuroscience and ∼80 MB for astronomy), the lack of overlap between shuffle and map stages results in Spark's slightly slower timings. This is a known limitation for Spark [39], and makes it slower than Dask and Myria for some of the steps in the context of image analytics as the shuffle cost for the larger image tuples is significant. Note that we tuned each of the systems to achieve the timings reported above. We discuss the impact of these tunings in Section 5.3.

5.2 Individual Step Performance

Next we focus on individual steps and examine performance across systems. Due to space restrictions we focus on the neuroscience pipeline.

Data Ingest: Input data for both use cases is staged in Amazon S3. For the neuroscience use case, image data is presented as a single NIfTI file per subject which contains compressed image volumes. For the neuroscience use case, the reference implementation works on all of the image volumes associated with the subject concurrently. To parallelize the implementation within each subject, we split data for each subject into separate image volumes represented by NumPy arrays, which can be processed in parallel. Therefore, NIfTI files need to be preprocessed for Spark and Myria, de-compressed and saved as serialized NumPy Arrays. SciDB requires NIfTI files to be converted into CSV format. TensorFlow requires images to be in NumPy array format for conversion to Tensors. Pre-processing times are included in data ingest times, which are shown in Figure 9. As the figure shows, data ingest times vary greatly across systems (note the log scale on the Y-axis). Spark and Myria download pre-processed data in parallel on each of the workers from S3. Myria is given a CSV list of images in S3 as part of the load statement, and Spark is given the S3 bucket name. Even though Myria's data ingest writes files to disk, it is faster than Spark, which loads the data into memory. This is because Spark enumerates the files in the S3 bucket on master before downloading them in parallel and meta-data querying in S3 is known to be a slow operation. For Dask, we manually specify the number of subjects to download per node, as otherwise Dask's scheduler assigns a random number of subjects to each node which lead to memory exhaustion or excessive data shuffle between processing steps. Thus, Dask's data ingest time looks like a step function: when the number of subjects is fewer than the number of nodes (16), each node downloads one subject concurrently. With more than 16 subjects, some nodes download two subjects. For the TensorFlow implementation, all data is downloaded to the master node and partitions are sent to each worker node. This is slower than the parallel ingest available in other systems. For SciDB, we report two sets of timings in Figure 9. SciDB-1 shows the time to ingest NumPy arrays with the `from_array()` interface, and SciDB-2 shows the time to convert NIfTI to CSV and ingest using the `aio_input` library. Because `aio_input()` reads in multiple files and parses them in parallel while SciDB's native Python API (i.e., scidb-py) processes input data in a serial manner, data ingest with the latter is an order of magnitude faster than the former and is on par with parallel ingest on Spark and Myria. Nevertheless, the NIfTI-to-CSV conversion overhead for SciDB is larger than the NIfTI-to-NumPy overhead for Spark and Myria, which makes SciDB ingest slower than Spark and Myria.

Segmentation (filtering): Segmentation is the first step in the neuroscience use case (i.e., Step ①$_N$). We discuss the performance of two operations in this step: filtering the data to select a subset of the image volumes, and computing an average image volume for each subject. Figure 11a and Figure 11b show the runtimes for these two operations as we vary the input data size on the 16-node cluster. Myria and Dask are the fastest on the data filtering step. Myria pushes the selection to PostgreSQL, which efficiently scans the data and returns the matching records (without indices) on the `Images` relation. Dask is fast on this operation as all data is in memory and the operation is a simple filter. Spark is an order of magnitude slower than Dask, even though data is in memory for both systems. This is because the filter criteria is specified as an anonymous function in Spark, and data and function have to be passed from Java to the external Python process and back. SciDB is slower than other systems because the internal chunks are not aligned with the selection predicate. In addition to scanning chunks, SciDB must also extract subsets of these chunks and construct new chunks in the resulting arrays. In TensorFlow, the data (tensors) takes the form of 4D arrays. For each subject, the 4D array represents the 288 3D data volumes. The selection is on the volume ID, which is the fourth dimension of the input data. However, TensorFlow only supports filtering along the first dimension. We thus need to flatten the 4D array, apply the selection, and reshape the array back into a 4D structure. As reshaping is expensive compared with filtering, TensorFlow is orders of magnitude slower than the other engines on this step.

Segmentation (mean): Figure 11b shows the result for the mean image volume computations. SciDB is the fastest for mean computation on the small datasets as it is designed to process arrays in parallel at the granularity of chunks. In contrast, Myria and Spark group data by subject, which leads to low cluster utilization for small numbers of subjects. The three systems have similar performance at larger scales. Dask is slower than the other three engines, especially for small datasets, due to startup and work stealing overheads. TensorFlow is very slow as the mean has to be computed in seperate graphs due to graph size limitations with data being sent to the master after each graph computation.

Denoising: Figure 11c shows the runtime for denoising (Step ②$_N$). For this step, the bulk of the processing happens in the user-defined denoising function. Dask, Myria, Spark, and SciDB all run the same code from the reference implementation on similarly partitioned data, which leads to similar overall performance. As in the case of the end-to-end pipeline, Dask's higher start-up overhead results in

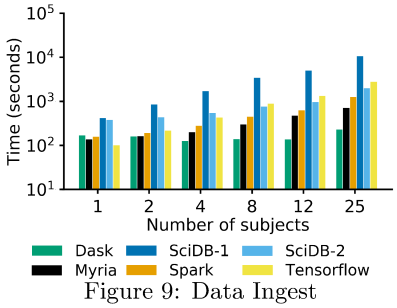


Figure 9: Data Ingest

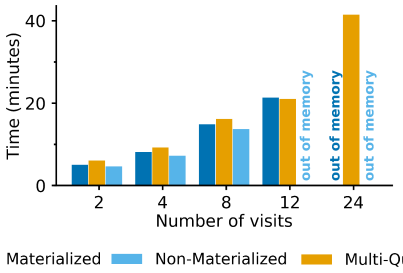


Figure 10: Variance in execution time, astronomy use case, Myria.

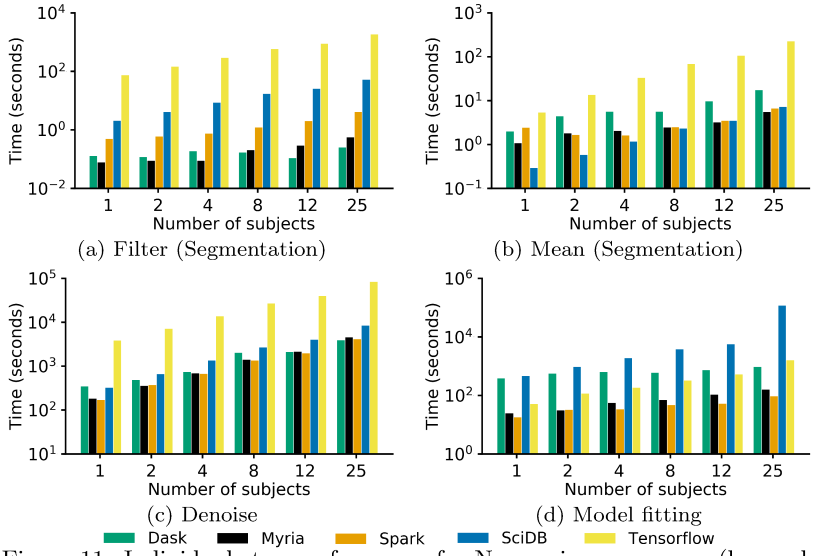


Figure 11: Individual step performance for Neuroscience use case (log scale on the y-axis). Experiments run on 16 nodes.

slightly worse performance for smaller data sizes, but similar performance for larger datasets. SciDB's `stream()` interface performs slightly worse than Myria, Spark, and Dask. For every call to the UDF we have to convert between string streams and the UDF's expected formats, this known overhead for SciDB is small for this step as more time is spent in computation compared to other steps. The TensorFlow implementation cannot use the binary mask to reduce the computation for each data volume. This is because TensorFlow's operations can only be applied to whole tensors and cannot be masked. This limitation and compute graph-size limitations contribute to slower performance of TensorFlow.

Model fitting: Figure 11d shows the runtime for the final step in the neuroscience pipeline. There are two important difference between denoising and model fitting. First, model fitting is less computationally intensive compared to denoising, Second, it offers a larger degree of data parallelism. For denoising, parallelism is limited to an image volume per subject (288 image volumes per subject). This is because all of the pixels from a single image volume are required to denoise an image volume. For model fitting, each voxel in each image can be processed independently, leading to $145 \times 175 \times 145$ potential data partitions per subject that can be processes in parallel. We implement model fitting on voxel batches rather than individual voxels to balance the cost of scheduling and benefits of parallelism. We use similar voxel batch sizes for all systems to ensure fair comparison ($\sim$500 partitions per subject). A higher degree of parallelism (more partitions) and smaller partition sizes (leading to smaller data shuffling times) for this step reduce Myria's pipelining advantage and make Spark faster. Dask is slower in this step as larger numbers of parallel computations lead to aggressive shuffling and job stealing, which dominate the processing time. We suspect that this is due to Dask's inaccurate estimate of the amount of data that needs to be shuffled among workers. This results in almost constant time for model fitting as the number of subjects increases. SciDB is the slowest. As the `stream()` interface does not support UDAs (only UDFs), the splitting and aggregation of the denoised images into voxel batches has to be done in AFL, which necessiates parsing the output from the previous step (i.e., denoising) from a stream of strings into SciDB arrays with the correct chunking schema, aggregating, and splitting into voxel batches. Finally, TensorFlow shows good performance on this step. Unlike denoising, which processe entire image volumes, model fitting executes on voxel batches, which can be filtered before the computation is performed. This and TensorFlow's efficient linear algebra implementation lead to a faster performance on this step compared to the other steps.

5.3 System Tuning

Finally, we evaluate the five systems on the complexity of the tuning required to achieve high performance.

Degree of Parallelism: This key parameter depends on three factors: (1) the number of machines in the cluster; (2) the number of workers that execute on each machine; and (3) the size of the data partitions that can be allocated to workers. We evaluated the impact of the cluster size in Section 5.1. Here, we evaluate the impact of (2) and (3).

For Myria, given a 16-node cluster, the degree of parallelism is entirely determined by the number of workers per node. As in traditional parallel DBMSs, Myria hash-partitions data across all workers by default. Figure 12 shows runtimes for different numbers of workers for the neuroscience use case. A larger number of workers yields a higher degree of parallelism but workers also compete for physical resources. For both use cases (astronomy not shown due to space constraints), four workers per node yields the best results. Changing the number of workers requires reshuffling or reingesting the data, which makes tuning this setting tedious and time-consuming.

Spark creates data partitions, which correspond to *tasks*, and each worker can execute as many tasks in parallel as available cores. The number of workers per node thus does not impact the degree of parallelism, as long as the number

Figure 12: 25 subjects, 16 nodes.

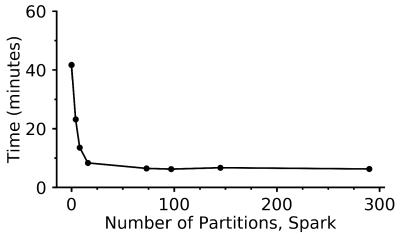


Figure 13: Single subject, 16 nodes.

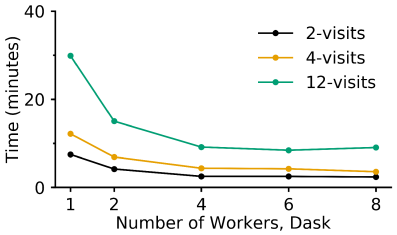


Figure 14: Multiple visits, 16 nodes.

of cores remains the same. The number of data partitions determines the number of tasks that can be scheduled, and the number of cores can be restricted in Spark to increase the memory available per core on each node. As Spark did not run out of memory for either use case we choose to utilize all the cores for the Spark implementation. We further discuss memory management later in this section.

Figure 13 shows the query runtimes for different numbers of input data partitions on Spark. On a 16-node cluster, the decrease in runtime was dramatic between 1 and 16 partitions, as an increasingly large fraction of the cluster became utilized. The runtime continues to improve until 128 data partitions, which is the total number of slots in the cluster (i.e., 16 nodes × 8 cores). Increasing the number of partitions from 16 to 97 resulted in 50% improvement. Further increases did not improve performance. Interestingly, if the number of data partitions is unspecified, Spark creates a partition for each HDFS block. The degree of parallelism then depends on the HDFS block size setting.

Dask allows specifying the number of workers per node and threads per worker. Through manual tuning, we find 8 to be the optimal number of workers per node when memory is abundant, as shown in Figure 14. Running multiple threads per process cause data corruption as the UDFs in Python are not all thread safe, so we use a single thread per process. Dask's work stealing automatically load balances across the machines, and work-stealing does not require any tuning.

In TensorFlow, we execute one process per machine. All steps include data conversions, which have to be performed on the master. These data conversions are the bottleneck, limiting opportunities for additional tuning. For all steps, we must partition the data such that graphs assigned to workers are below 2GB in size. Additionally, for the denoising step memory is the bottleneck requiring the assignment of only one image volume at a time per physical machine. The model fitting step can be parallelized at the granularity of individual voxels and we find that TensorFlow performs best with 128 partitions on a 16-worker configuration. In general, we observe that TensorFlow tends to perform better with smaller chunk sizes.

The SciDB documentation recommends [37] running one instance per 1-2 CPU cores. The chunk size, however, is more difficult to tune: we do not find a strong correlation between the overall performance and common system configurations. We tune the chunk size for each operation by running the application with the same data but using different chunk sizes. For instance, in Step ③$_A$ we find that a chunk size of [1000 × 1000] leads to the best performance. A chunk size of [500 × 500], for example, is 3× slower; while chunk sizes of [1500 × 1500] and [2000 × 2000] are 22% and 55% slower, respectively.

Memory Management: Image analytics workloads are memory intensive. Additionally, data allocation can be skewed across compute nodes. For example, the astronomy pipeline grows the data by 2.5× on average during processing, but some workers experience data growth of 6× due to skew. As a result, systems can easily run out of memory. Big data systems use different approaches to trade off query execution time with memory consumption. We evaluate some of these trade-offs in this section. In particular, we compare the performance of pipelined and materialized query execution. With pipelined execution, data flows directly from one operator to the next without going to disk and without synchronization barrier. Materialized query execution in contrast writes the output of each operation to disk and the next operation starts by reading input data from disk. For materialized query execution, we compare materializing intermediate results and processing subsets of the input data at a time. Figure 10 shows the results for the Myria system on the astronomy use case. As the figure shows, when data is small compared with the available memory, the fastest way to execute the query is to pipeline intermediate results from one operator to the next. This approach is 8-11% faster in this experiment than materialization, and 15-23% faster than executing multiple queries. As the ratio of data-to-memory grows, pipelining may cause a query to fail with an out-of-memory error. Intermediate result materialization then becomes the fastest query execution method. With even more data, the system must cut the data analysis into even smaller pieces for execution without failure.

In contrast to Myria, Spark automatically spills intermediate results to disk to avoid out-of-memory failures. Additionally, Spark can partition data into smaller fragments than the number of available nodes, and will process only as many fragments as there are available slots. Both techniques help to avoid out-of-memory failures with Spark. However, the lack of pipelined execution causes Spark to be slower than Myria when memory is plentiful (see also Figure 8g).

Dask supports spilling to disk, but the current implementation of this feature is not multi-process safe and thus not suitable for our use cases. To prevent running out of memory in Dask, we increased the memory-to-CPU ratio by reducing the number of workers on each node. For the 32-node cluster, we reduced the number of workers to one worker per node in the astronomy use case. On a 16-node cluster reducing the number of workers to one did not help and the use case could not run to completion. This was due to skew rather than insufficient memory: 2 workers in the astronomy use case ran out of memory and caused cascading failures.

6. SUMMARY AND FUTURE WORK

In this section, we summarize the lessons learned from three perspectives: system developers, users, and researchers:

Developers. Engine developers can improve both the architecture and implementation of their systems based on our observations, some of which are already known, but their importance is re-emphasized by this study. Most importantly, we find that all evaluated systems would benefit from automatically adjusting the degree of parallelism and gracefully spilling to disk, even when individual data partitions do not fit in memory to avoid all sources of out-of-memory failures.

Image analytics differs from other analytics in three respects: the large size of individual records (i.e., image fragments with metadata), heavy use of UDFs to execute complex, domain-specific operations, and the multidimensional nature of the data. Some systems, such as SciDB, have only limited support for UDFs/UDAs in languages other than C++. As we showed in Section 5, this significantly affects performance and ease of use. In contrast, only SciDB reasons about multidimensional array data. In all other systems, users must manually split images into fragments along different dimensions in preparation for their analysis, which is non-trivial.

Finally, most systems are optimized for large numbers of small records rather than small numbers of large records. Myria, for example, processes tuples in large batches by default. We had to change that default to reduce the number of tuples per batch and prevent out of memory failures.

We next discuss specific recommendations for each of the evaluated systems for running image analytical workloads.
Dask would further benefit from (1) a simpler API: e.g., reduce the number of ways to construct the compute graph (2) better debuggability as noted in Section 4.1; and (3) spilling to disk for multi-process workloads as noted in Section 5.3.
Myria would benefit from (1) automatically tuning the number of workers per machine and making it easier to change the number of workers as noted in Section 5.3; (2) adding support for local combiners before shuffles for user-defined aggregations: this would lead to fewer memory issues in case of skew. We were able to materialize intermediate results and split queries into multiple ones to achieve the same result, but it required better understanding of Myria and more effort.
Spark would benefit from (1) overlapping the shuffle phase with the map phase to increase performance when memory is sufficient and (2) making parallel data ingest from S3 more efficient.
SciDB would benefit from (1) binary data format support for the `aio_input()` interface; (2) support for more than TSV and stdin-stdout for the `stream()` interface; (3) more efficient methods for concatenating arrays; (4) support for advanced control over the child process such as setting environment variables; (5) simplified procedure for multi-node deployment; and (6) support for UDAs in the stream interface.
TensorFlow would benefit from (1) removing the restriction on graph size; (2) better tooling for cluster management and scheduling; (3) distributed data ingest; and (4) support for external libraries.
Users. For domain scientists wanting to utilize big data systems there are several considerations: (1) Re-write or re-use: can the computation be expressed in native SQL or AQL? If the computation is simple this may be the most performant solution. If not, systems such as Dask, Spark, and Myria can efficiently execute legacy Python (or other) scripts with minimal additional code provided by the user. (2) Data partitioning: turning a serial computation into a parallel one may pose the biggest challenge to domain users. A reference implementation may or may not indicate how computation can be parallelized. Understanding data dependency and the synchronization points in underlying computation is crucial to ensuing correctness and performance in a big data system.
Researchers. Our study raises a number of research questions. Image processing involves complex analytics, which include iterations and linear algebra operations that must be efficiently supported in big data systems. However, users typically have legacy code that performs sophisticated and difficult to rewrite operations. Therefore, they need the ability to call existing libraries. They also need an easy mechanism to parallelize the computation: they should be able to reason about multidimensional array data directly rather than manually creating and processing collections of image fragments. It should be easy to mix and match UDF/UDA computations and pre-defined (e.g., relational) operations on complex data such as image fragments.

Our study also re-iterates the general need to efficiently support pipelines with UDF/UDAs both during query execution and query optimization. Image analytics implies large tuples and larger tuples put pressure on memory management techniques, systems' ability to shuffle data efficiently, and efficient methods to pass large records back and forth between core computation and UDFs/UDAs. This provides another research opportunity. Finally, making big data systems usable for scientists requires systems to be self tuning, which is already an active research area [19].

7. RELATED WORK

Traditionally, image processing research has focused on effective indexing and querying of multi-media content [14, 11, 12]. Typical DBMS benchmarks (e.g., [43]) focus on business intelligence computations over structured data. The GenBase benchmark [41] takes this forward to focus on complex analytics, but does not examine image data. Several recent papers [32, 27, 8, 39] evaluate the performance of Big Data systems, but the workload does not include image analysis. While prior work on raw files and scientific formats [4, 9] focuses on techniques for working with them directly, it does not offer mechanisms to work with them in big data systems like the ones evaluated in this paper.

8. CONCLUSION

We presented the first comprehensive study of large-scale image analytics on big data systems. We surveyed the different paradigms of large-scale data processing platforms using two real-world use cases from domain sciences. While we could execute the use cases on these systems, our analysis shows that leveraging the benefits of all systems requires deep technical expertise. For these systems to better support image analytics in domain sciences, they need to simultaneously provide comprehensive support for multidimensional data and high performance for UDFs/UDAs written in popular languages (e.g., Python). Additionally, they need to completely automate data and compute distribution across a cluster and memory management to eliminate all possible sources of out-of-memory failures. Overall, we argue that current systems provide

good support for image analytics, but they also open new opportunities for further improvement and future research.

Acknowledgements: This project is supported in part by the National Science Foundation through NSF grants IIS-1247469, IIS-1110370, IIS-1546083, AST-1409547 and CNS-1563788; DARPA award FA8750-16-2-0032; DOE award DE-SC0016260, DE-SC0011635, and from the DIRAC Institute, gifts from Amazon, Adobe, Google, the Intel Science and Technology Center for Big Data, award from the Gordon and Betty Moore Foundation and the Alfred P Sloan Foundation, and the Washington Research Foundation Fund for Innovation in Data-Intensive Discovery. Human MRI data were provided by the Human Connectome Project, WU-Minn Consortium (Principal Investigators: David Van Essen and Kamil Ugurbil; 1U54MH091657) funded by the 16 NIH Institutes and Centers that support the NIH Blueprint for Neuroscience Research; and by the McDonnell Center for Systems Neuroscience at Washington University.

9. REFERENCES

[1] http://dask.pydata.org.
[2] M. Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems. In *OSDI*, 2016.
[3] http://abcdstudy.org/.
[4] I. Alagiannis et al. Nodb: efficient query execution on raw data files. In *SIGMOD*, 2012.
[5] https://aws.amazon.com/.
[6] P. J. Basser et al. Estimation of the effective self-diffusion tensor from the NMR spin echo. *J. Magn. Reson. B*, 1994.
[7] https://bazel.build/.
[8] M. Bertoni et al. Evaluating cloud frameworks on genomic applications. In *IEEE International Conference on Big Data*, 2015.
[9] S. Blanas et al. Parallel data analysis directly on scientific file formats. In *SIGMOD*, 2014.
[10] P. G. Brown. Overview of scidb: Large scale array storage, processing and analysis. In *SIGMOD*, 2010.
[11] C. Carson et al. Blobworld: A system for region-based image indexing and retrieval. In *VISUAL*, 1999.
[12] S. Chaudhuri et al. Optimizing top-k selection queries over multimedia repositories. *IEEE Trans. Knowl. Data Eng.*, 2004.
[13] P. Coupe et al. An optimized blockwise nonlocal means denoising filter for 3-D magnetic resonance images. *IEEE Trans. Med. Imaging*, 27(4), Apr. 2008.
[14] C. Faloutsos et al. Efficient and effective querying by image content. *J. Intell. Inf. Syst.*, 1994.
[15] http://fits.gsfc.nasa.gov.
[16] https://www.microsoft.com/en-us/research/publication/fourth-paradigm-data-intensive-scientific-discovery.
[17] E. Garyfallidis et al. Dipy, a library for the analysis of diffusion MRI data. *Front. Neuroinform.*, 8, 21 Feb. 2014.
[18] D. Halperin et al. Demonstration of the myria big data management service. In *SIGMOD*, 2014.
[19] H. Herodotou et al. Starfish: A self-tuning system for big data analytics. In *CIDR*, 2011.
[20] http://astroinf.cmm.uchile.cl/category/projects/.
[21] https://medium.com/data-collective/rapid-growth-in-available-data-c5e2705a2423.
[22] http://blog.d8a.com/post/9662265140/the-growth-of-image-data-mobile-and-web.
[23] T. L. Jernigan et al. The pediatric imaging, neurocognition, and genetics (ping) data repository. *Neuroimage*, 2016.
[24] N. Ji, , et al. Technologies for imaging neural activity in large volumes. *Nat. Neurosci.*, 2016.
[25] https://www.lsst.org/.
[26] http://dm.lsst.org/.
[27] O. Marcu et al. Spark versus flink: Understanding performance in big data analytics frameworks. In *IEEE (ICCC)*, 2016.
[28] K. L. Miller et al. Multimodal population brain imaging in the uk biobank prospective epidemiological study. *Nature Neuroscience*, 2016.
[29] https://nifti.nimh.nih.gov/nifti-1.
[30] T. E. Oliphant. Python for scientific computing. *Computing in Science and Engg.*, 2007.
[31] N. Otsu. A threshold selection method from gray-level histograms. *Automatica*, 1975.
[32] A. Pavlo et al. A comparison of approaches to large-scale data analysis. In *SIGMOD*, 2009.
[33] https://www.postgresql.org/.
[34] http://www.rasdaman.org/.
[35] M. Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference*, 2015.
[36] http://www.scidb.org/.
[37] http://forum.paradigm4.com/t/persistenting-data-to-remote-nodes/1408/8.
[38] http://skyserver.sdss.org/dr7.
[39] J. Shi et al. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *PVLDB*, 2015.
[40] http://spark.apache.org/.
[41] R. Taft et al. Genbase: a complex analytics genomics benchmark. In *SIGMOD*, 2014.
[42] The Dark Energy Survey Collaboration. The Dark Energy Survey. *ArXiv Astrophysics e-prints*, Oct. 2005.
[43] http://www.tpc.org.
[44] D. C. Van Essen et al. The WU-Minn human connectome project: an overview. *Neuroimage*, 15 Oct. 2013.
[45] B. A. Wandell. Clarifying human white matter. *Annu. Rev. Neurosci.*, 1 Apr. 2016.
[46] J. Wang et al. The myria big data management and analytics system and cloud services. In *CIDR*, 2017.
[47] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.