Image Analysis At Scale: 
A Comparison of Five Systems

Jake VanderPlas @jakevdp
SciPy 2017, July 13 2017

Slides at:
http://speakerdeck.com/jakevdp/image-analysis-at-scale/
Image Analysis At Scale: A Comparison of Five Systems

Jake VanderPlas @jakevdp
SciPy 2017, July 13 2017

Parmita Mehta, Sven Dorkenwald, Dongfang Zhao, Tomer Kaftan, Alvin Cheung, Magdalena Balazinska, Ariel Rokem, Andrew Connolly, Jake VanderPlas, Yusra AlSayyad
Comparative Evaluation of Big-Data Systems on Scientific Image Analytics Workloads

Experiments and Analysis

Parmita Mehta, Sven Dorkenwald, Dongfang Zhao, Tomer Kaftan, Alvin Cheung
Magdalena Balazinska, Ariel Rokem, Andrew Connelly, Jacob Vanderplas, Yusra AlSayyad
University of Washington

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, however, how well these systems support real-world image analysis use cases, and how performant are the image analytics tools implemented on top of such systems. 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.

For example, the UK biobank will release Magnetic Resonance Imaging (MRI) data from close to 500k human brains (more than 200 TB) for neuroscientists to analyze [24]. Multiple other initiatives are similarly making large collections of image data available to researchers [2, 20, 37].

Such use cases emphasize the need for effective tools to support the management and analysis of image data: tools that are efficient, scale well, and are easy to program without requiring deep systems expertise to deploy and tune.

Surprisingly, there has been only limited work from the data management research community in building tools to support large-scale image management and analytics. Rassdaman [30] and SciDB [32] are two well-known DBMSs that specialize in the storage and processing of multidimensional array data and they are a natural choice for implementing image analytics. Most other work developed for storing image data targets pre-

How to Write a CS Paper . . .
How to Write a CS Paper . . .

1. Find a well-defined computing problem

“Efficient generation of Fibonacci numbers is a perennial problem in Computer Science, and no agreed-upon standard solution yet exists.”
How to Write a CS Paper . . .

1. Find a well-defined computing problem

   “Efficient generation of Fibonacci numbers is a perennial problem in Computer Science, and no agreed-upon standard solution yet exists.”

2. Design a tool that solves that problem efficiently

   “We present FibDB, the first ever relational database specifically designed for the generation and storage of numbers in the Fibonacci sequence.”
How to Write a CS Paper . . .

1. Find a well-defined computing problem

   “Efficient generation of Fibonacci numbers is a perennial problem in Computer Science, and no agreed-upon standard solution yet exists.”

2. Design a tool that solves that problem efficiently

   “We present FibDB, the first ever relational database specifically designed for the generation and storage of numbers in the Fibonacci sequence.”

3. Show that it's 1000x faster than Hadoop.
How to Write a CS Paper . . .

1. Find a well-defined computing problem

   “Efficient generation of Fibonacci numbers is a perennial problem in Computer Science, and no agreed-upon standard solution yet exists.”

2. Design a tool that solves that problem efficiently

   “We present FibDB, the first ever relational database specifically designed for the generation and storage of numbers in the Fibonacci sequence.”

3. Show that it’s 1000x faster than Hadoop.
   Use a bar chart. With log scales.
How to Write a CS Paper . . .

1. Find a well-defined computing problem

   “Efficient generation of Fibonacci numbers is a perennial problem in Computer Science, and no agreed-upon standard solution yet exists.”

2. Design a tool that solves that problem efficiently

   “We present FibDB, the first ever relational database specifically designed for the generation and storage of numbers in the Fibonacci sequence.”

3. Show that it’s 1000x faster than Hadoop.
   Use a bar chart. With log scales.

4. Repeat until tenured.
(I'm so sorry)
Paper Goal: evaluate existing Big Data systems on real-world scientific image analysis workflows & point the way forward for database & systems researchers.

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

Experiments and Analysis

Parmita Mehta, Sven Dorkenwald, Dongfang Zhao, Tomer Kaftan, Alvin Cheung
Magdalena Balazinska, Ariel Rokem, Andrew Connolly, Jacob Vanderplas, Yusra AlSayyad
University of Washington

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, however, how well these systems support real-world image analysis use cases, and how performant are the image analytics tasks implemented on top of such systems. 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.

For example, the UK biobank will release Magnetic Resonance Imaging (MRI) data from close to 500k human brains (more than 200 TB) for neuroscientists to analyze [24]. Multiple other initiatives are similarly making large collections of image data available to researchers [2, 20, 37].

Such use cases emphasize the need for effective tools to support the management and analysis of image data: tools that are efficient, scale well, and are easy to program without requiring deep systems expertise to deploy and tune.

Surprisingly, there has been only limited work from the data management research community in building tools to support large-scale image management and analytics. Radaman [30] and SciDB [32] are two well-known DBMSs that specialize in the storage and processing of multidimensional array data and they are a natural choice for implementing image analytics. Most other work developed for storing image data targets pre-
Goals of This Talk:

Distill lessons learned for the SciPy audience, which is largely made up of scientific practitioners.

Give a general idea of the strengths and weaknesses of each system, and what you might expect if applying it to your own research task.
Challenges for Scaling Scientific Image Analysis

1. Individual images are BIG, and typical databases aren't optimized for very large data units.
2. Images generally stored in domain-specific formats (FITS, NIfTI-1, etc.)
3. Requires specialized operations (e.g. filtering, aggregations, slicing, stencils, spatial joins)
4. Requires specialized analytics (e.g. background estimation, source detection, model fitting)
Neuroscience Case Study

Step 1: Segmentation
Separate foreground from background using Otsu segmentation algorithm
Neuroscience Case Study

*Step 1: Segmentation*
Separate foreground from background using Otsu segmentation algorithm

*Step 2: Denoising*
Use a local means filter to remove noise from images
Neuroscience Case Study

**Step 1: Segmentation**
Separate foreground from background using Otsu segmentation algorithm

**Step 2: Denoising**
Use a local means filter to remove noise from images

**Step 3: Model Fitting**
Fit a tensor model to describe diffusion within each voxel
**Five Systems:**

<table>
<thead>
<tr>
<th>System</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SciDB</td>
<td>Database architecture purpose-built for computation on multi-dim arrays.</td>
</tr>
<tr>
<td>Apache Spark</td>
<td>Popular in-memory big data system with wide adoption &amp; Python interface</td>
</tr>
<tr>
<td>Myria</td>
<td>Shared-nothing DBMS developed by members of our UW team</td>
</tr>
<tr>
<td>Dask</td>
<td>Python package aimed at parallelization of scientific workflows</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>System optimized for operations on N-dimensional tensors.</td>
</tr>
</tbody>
</table>
from scidbpy import connect
sdb = connect(url="...")
data_sdb = sdb.from_array(data)
data_filtered = data_sdb.compress(
    sdb.from_array(gtab.b0s_mask),
    axis=3)  # Filter
mean_b0_sdb = data_filtered.mean(index=3)  # Mean

Language: AQL/AFL or NumPy-like Syntax
UDFs*: Python UDF support via stream() interface
Data: Ingested as CSV, passed around pipelines as TSV

*UDF = “User Defined Function”
Advantages:
- Efficient native support for dense arrays & common operations (windows, joins, etc.)
- Python UDFs supported via `stream()` interface

Challenges:
- Data passed to UDFs in TSV format, leading to significant data transformation overhead in the pipeline
- Difficult installation process, no good support for cloud deployment
- Integration with external packages (e.g. LSST stack) is quite difficult
- `stream()` I/O read through stdin/stdout only, which breaks if the UDF uses this for other purposes
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)

**Language:** functional programming API

**UDFs:** Built-in support for Python UDFs

**Data:** Spark-specific RDDs (Resilient Distributed Datasets)
Advantages:
- Arbitrary Python objects as keys & straightforward Python UDFs streamlined implementation
- Succinct functional programming interface written in Python
- Large user community and extensive documentation

Challenges:
- Caching of intermediate results is not automatic, which can lead to silent repeated computation
- Initial implementation easy, but required extensive tuning to attain computational efficiency
conn = MyriaConnection(url="...")
conn.create_function("Denoise", Denoise)
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
                 PYUDF(Denoise, T1.img, T1.mask) as img,
                 T1.subjId, T1.imgId];
""

Language: MyriaL hybrid declarative/imperative language
UDFs: Built-in support for Python UDFs
Data: Flexible BLOB format (here: NumPy arrays)
Advantages:
- Can directly leverage existing Python implementations
- Declarative/Imperative MyriaL syntax is more flexible than typical DB languages (e.g. easily supports iteration)

Challenges:
- Greatest efficiency attained by reimplementation of key pieces of the algorithm
- Initial implementation easy, but required extensive tuning to attain computational efficiency
for id in subjectIds:
    data[id].vols = delayed(downloadAndFilter)(id)

for id in subjectIds:  # barrier data
    [id].numVols = len(data[id].vols.result())

for id in subjectIds:
    means = [delayed(mean)(block) for block in partitionVoxels(data[id].vols)]
    means = delayed(reassemble)(means)
    mask = delayed(median_otsu)(means)

**Language:** Pure Python  
**UDFs:** Supported via delayed(XXX)  
**Data:** anything Python can handle
Advantages:
- Simplest installation & deployment
- Python from the ground-up with familiar interfaces
- Built-in Python UDFs: required little re-implementation of algorithms

Challenges:
- User must reason about when to insert evaluation barriers in graphs
- User must choose manually how data should be partitioned across nodes
- Options like futures and delayed make Dask flexible, but somewhat harder to use.
- Difficult to debug: failed tasks go to a no-worker queue & can cause deadlock
```python
pl_inputs = []
work = []
for i_worker in range(len(steps[0])):
    with tf.device(steps[0][i_worker]):
        pl_inputs.append(tf.placeholder(shape=sh))
        work.append(tf.reduce_mean(pl_inputs[-1]))
mean_data = []
```

**Language:** Python used to manually set up workers  
**UDFs:** Not supported  
**Data:** TF-specific data structures, must be loaded on master node & distributed manually
Advantages:

Challenges:
- Limited support for distributed computation – user must manually map data & computation to workers
- 2GB serialized graph size limit means pipeline had to be manually broken into smaller steps
- Lack of Python UDFs requires complete re-implementation of algorithm using tensorflow primitives
- Limited set of built-in operations (e.g. does not support element-wise data assignment)

(It's clear that we are attempting to push tensorflow well beyond its design goals. It's still an excellent tool for what it was designed for, namely deep learning workflows)
- Dask/Myria/Spark: similar performance, as they are all essentially distributing the same Python UDFs

- SciDB: slower primarily due to conversion of data to/from TSV at the input/output of each Python UDF

- Tensorflow: slower due to many limitations previously discussed
See our paper for more detailed quantitative breakdown & discussion

https://arxiv.org/abs/1612.02485
### Key Takeaways:

<table>
<thead>
<tr>
<th>Dask</th>
<th>Myria</th>
<th>SciDB</th>
<th>Spark</th>
<th>Tensorflow</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Dask
- Myria
- SciDB
- Spark
- Tensorflow
**Key Takeaways:**

Scientific pipelines are complex enough that they rarely map onto built-in primitives for existing big data systems.

<table>
<thead>
<tr>
<th>Sufficient Primitives</th>
<th>Dask</th>
<th>Myria</th>
<th>SciDB</th>
<th>Spark</th>
<th>Tensorflow</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>


N/A
Key Takeaways:

In the meantime, seamless support for user-defined functions (UDFs) is absolutely **essential** for scientific use-cases.

<table>
<thead>
<tr>
<th></th>
<th>Dask</th>
<th>Myria</th>
<th>SciDB</th>
<th>Spark</th>
<th>Tensorflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sufficient Primitives</td>
<td>N/A</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Python UDF Support</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✗</td>
</tr>
</tbody>
</table>
**Key Takeaways:**

<table>
<thead>
<tr>
<th></th>
<th>Dask</th>
<th>Myria</th>
<th>SciDB</th>
<th>Spark</th>
<th>Tensorflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sufficient Primitives</td>
<td>N/A</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Python UDF Support</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Flexible data formats</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>

Support for flexible domain-specific data formats in pipelines is very important for any nontrivial computational task.
### Key Takeaways:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Dask</th>
<th>Myria</th>
<th>SciDB</th>
<th>Spark</th>
<th>Tensorflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sufficient Primitives</td>
<td>N/A</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Python UDF Support</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Flexible data formats</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Automatic tuning</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Ideally, parallel computations & memory usage should be tuned automatically by the systems. None of the explored systems do this particularly well.
### Key Takeaways:

<table>
<thead>
<tr>
<th></th>
<th>Dask</th>
<th>Myria</th>
<th>SciDB</th>
<th>Spark</th>
<th>Tensorflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sufficient Primitives</td>
<td>N/A</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Python UDF Support</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Flexible data formats</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Automatic tuning</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Streamlined Installation</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Installation headaches are the easiest way to drive frustration. Streamlined installation, particularly on the cloud, is a must.
### Key Takeaways:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Dask</th>
<th>Myria</th>
<th>SciDB</th>
<th>Spark</th>
<th>Tensorflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sufficient Primitives</td>
<td>N/A</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Python UDF Support</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>Flexible data formats</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>Automatic tuning</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Streamlined Installation</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Large User Community</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

A large and active user & developer community makes solving problems & getting questions answered much easier.
Who wins?

Lack of primitives means each is an exercise in sending Python UDFs to data on distributed nodes.

This is an ancillary mode of computation for most systems, and skips many of their efficiencies.

Exception is Dask, which is specifically designed for this mode of computation.

Bottom Line: Use Dask unless you know your use-case is covered by other systems’ primitives.
Thank You!

Email: jakevdp@uw.edu
Twitter: @jakevdp
Github: jakevdp
Web: http://vanderplas.com/
Blog: http://jakevdp.github.io/

Slides: http://speakerdeck.com/jakevdp/image-analysis-at-scale/

Associated code is in a private GitLab repository and will be released after VLDB in August
Key Takeaway:

Existing big data systems have many potential areas of improvement for supporting scientific workflows.

We hope our paper will point the way for researchers developing these systems.
Extra Slides
Case Studies:

Neuro-Imaging

Human Connectome Project
900 subjects x 288 3D dMRI
“images”, 145 x 145 x 174 voxels
Total size: 105GB, NIfTI-1 format

Tasks:
- Segmentation & Masking
- Denoising
- Model Fitting
Case Studies:

Neuro-Imaging

Human Connectome Project
900 subjects x 288 3D dMRI “images”, 145 x 145 x 174 voxels
Total size: 105GB, NIfTI-1 format
Tasks:
- Segmentation & Masking
- Denoising
- Model Fitting

Astronomy

High Cadence Transient Survey
24 Visits x 60 2D Images + noise estimates, 4000 x 4072 pixels
Total size: 115GB, FITS format
Tasks:
- Pre-processing & Cleaning
- Patch creation
- Co-addition
- Source detection
Evaluation:

Qualitative:
- How easy is it to implement scientific pipelines?
- Can existing pipelines run on the system?
- How much effort is required to implement?
- How much technical expertise is required to optimize the system?

Quantitative:
- What is the memory consumption?
- What is the end-to-end runtime?
- What is the runtime for each implemented step?
Neuroscience: Data Ingest

SciDB 1: data ingest via NumPy array
SciDB 2: data ingest direct from CSV
Neuroscience: Filter and Mean

![Graphs showing time(seconds) vs number of subjects for different tools: Dask, SciDB, Myria, Spark, and Tensorflow.]
Neuroscience: Denoise and Model Fit

![Comparison of Dask, SciDB, Myria, Spark, and Tensorflow for different numbers of subjects](image)
Astro Pipeline

![Graph showing time in minutes vs. number of visits for Materialized, Non-Materialized, and Multi-Query categories.](image-url)
Lessons Learned for Developers

Scientific image analytics requires:
- Easy manipulation of multidimensional array data
- Processing with sophisticated UDFs and UDAs

More generally:
- Make systems easy to deploy and easy to debug
- Automatically tune degree of parallelism and other configuration parameters
- Gracefully spill to disk: out-of-memory errors remain too common
- Read existing, scientific file formats
Lessons Learned for Users

Key Decision: Reuse or Rewrite
- Rewriting code can yield higher performance
- Reusing saves time and avoids new bugs

Turning a serial computation into a parallel computation remains challenging
Lessons Learned for Researchers

Need to efficiently support pipelines with UDFs

Image analytics is memory intensive
- Need to efficiently manage memory
- Individual records are large

Self-tuning & robust systems are a must.