Data Pipelines with Pachyderm

David A. Ventimiglia

<2024-07-16 Tue>

Data pipelines are hard.

We keep attempting to solve this problem.

AWS Data Pipeline S3, EC2, EMR, ...

Apache Airflow DAGS, Tasks, Schedulers, ...

Pachyderm Pipelines, Repositories, Jobs, ...

Data pipelines are necessary.

Chegg has loads of them, mostly using Apache Spark and databricks.

- Qualify UGC flash-card decks for SEO.
- Generate embeddings for similarity search.
- Classify UGC images for quality, up-scaling, and segmentation.

Distributed systems are hard.

GNU Make Distribute recipes across cores.

GNU Parallel Distribute Bash pipelines across cores.

dask Distribute data structures across threads, cores, machines.

Pachyderm distribute *"datum" transformations* across pods.

Pachyderm distributes "datum" transformations across pods.

Features Data-driven pipelines, Version Control, Auto-scaling & Deduplication

Concepts PFS, Repos, Branches, Commits, Pipelines, Jobs, Datums, Projects

Data-driven pipelines

Automatically trigger pipelines based on changes in the data and only process dependent changes.

Version Control

Automatically track changes to any file with Git semantics.

Auto-scaling & Deduplication

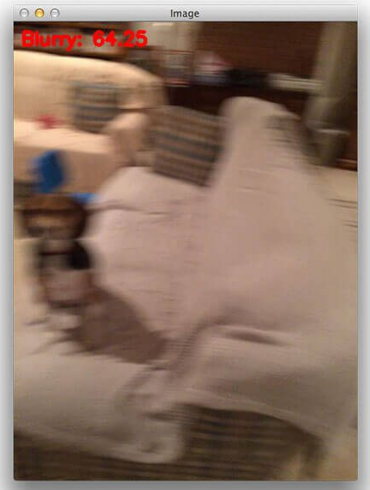
Automatically scale jobs, parallelize data sets, and deduplicate across repositories.

UGC image quality is important.

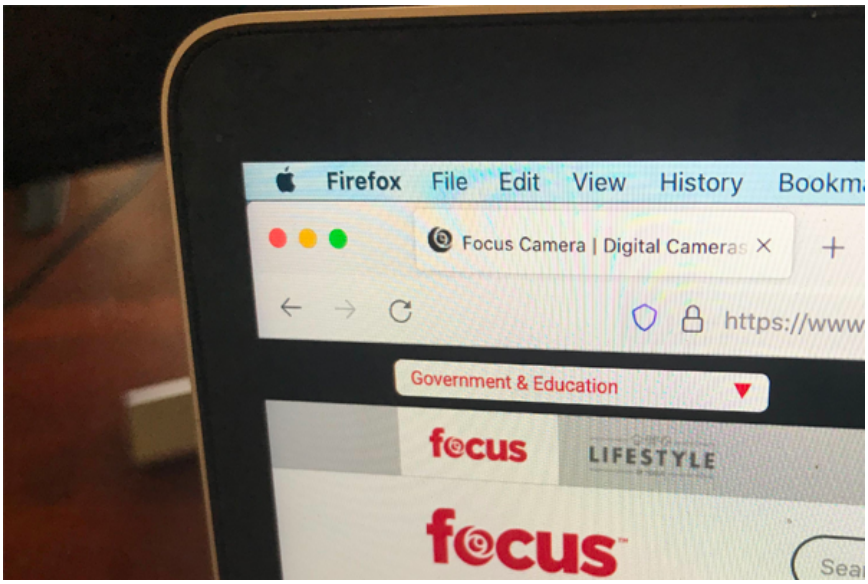
blur variance-of-Laplacian and Sobel filters

Moiré Fourier transform

Blur



Moiré



Variance-Of-Laplacian

Convolve greyscale image with Laplacian 3x3 kernel and take the variance.

```
import cv2
def vol(f):
    img = cv2.imread(f)
    grey = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    return cv2.Laplacian(grey, cv2.CV_64F).var()
```

Sobel Filters

Compute approximate gradient of a greyscale image in the X and Y coordinates and take the variance.

```
import cv2
def sob(f):
    img = cv2.imread('/tmp/ugc.jpg')
    grey = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    return cv2.Sobel(grey, cv2.CV_64F, 0, 1).var()
```

Fourier Transform

Convert image to frequency-domain, remove low-frequencies, apply inverse transform, return the mean of the magnitude.

```
import cv2
import numpy as np
def fft(f):
    img = cv2.imread('/tmp/ugc.jpg')
    grey = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    img_fft = np.fft.fft2(grey)
    ...
    recon = np.fft.ifft2(fftShift)
    magnitude = 20 * np.log(np.abs(recon))
    mean = np.mean(magnitude)
    return mean
```

Naive Bayes Classifier

Simple linear classifier based on the likelihood of events *assumed* to be independent.

```
from sklearn import svm
from sklearn.naive_bayes import GaussianNB
```

```
clf = GaussianNB()
def cls(x):
    y = clf.predict(x)
    return clf.predict(x)
```

Chegg Image Quality & Pachyderm.

- prototyped a series of Python OpenCV scripts
- orchestrated on a single workstation with GNU Make
- deploy it at scale and with k8s expertise
- deploy it in an online setting (SPOILER ALERT! That didn't happen!)
- try something different

Prototyping locally with Python and Make

Scripting

Python script to read a filename from the command line args, load the image, compute metrics, and write it to a file with another name from the args.

```
import sys
with open(sys.argv[2], 'w') as f:
    f.write(vol(sys.argv[3]))
```

Scripting

Python script to read multiple filenames from command line args, compute the classification, and write it out to a file with another name from the args.

```
import sys
with open(sys.argv[3], 'w') as vol, open(sys.argv[4], 'w') as sob, open(sys.argv[5], 'w
      cls.write(cls.predict([
                float(vol.read()),
                float(sob.read()),
                float(fft.read())]))
```

Orchestrating

GNU Make script using pattern matching and parallel execution with `.jpg` prerequisites, OpenCV outputs as targets, and Python scripts as recipes

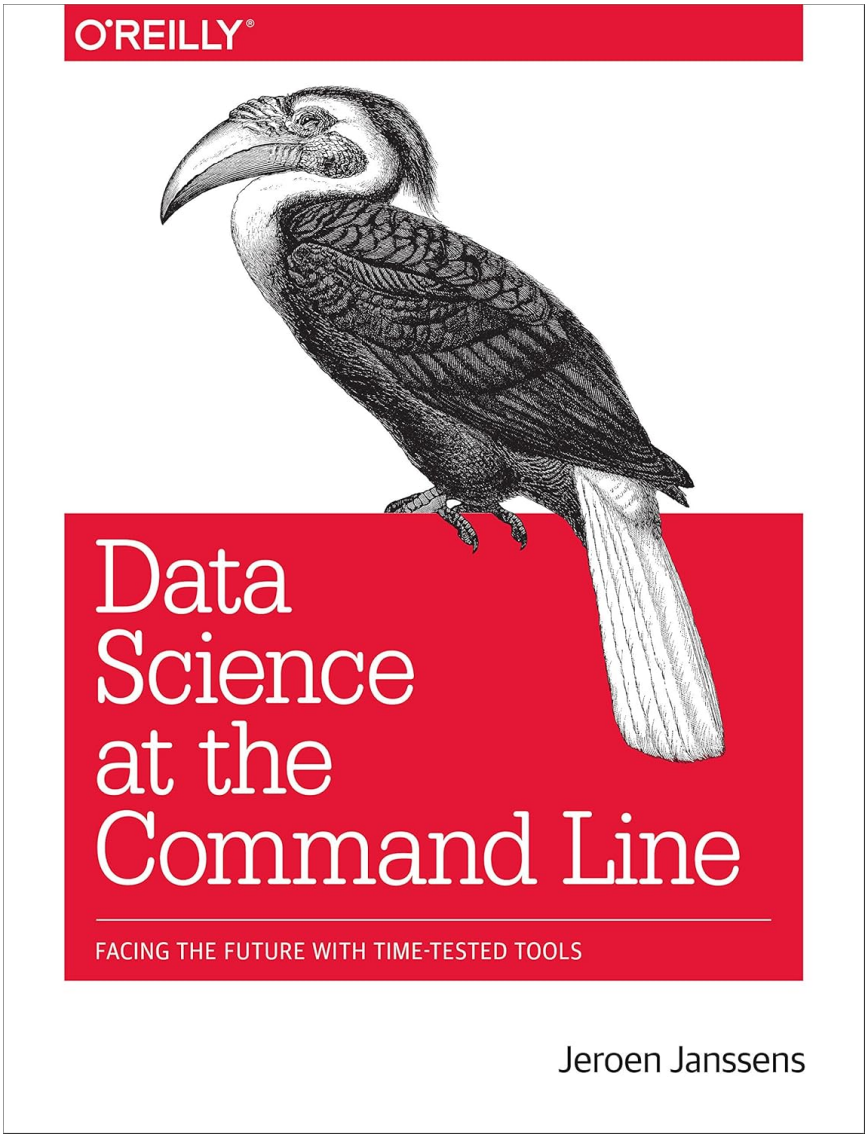
```
TARGETS := $(patsubst %.jpg,%.cls,$(wildcard *.jpg))
all: $(TARGETS)
$(TARGETS): %.cls:%.vol %.cls:%.sob %.cls:%.fft
        python cls.py $< $@
%.vol: %.jpg
        python vol.py $< $@
%.sob: %.jpg
        python sob.py $< $@
%.fft: %.jpg
        python fft.py $< $@

make -j
```

Deploying at scale using Pachyderm

Scripting

Use the exact same Python scripts since the Pachyderm execution model reads files from the the PFS and write files to the PFS filesystem.



Orchestrating

Pachyderm has "pipelines" defined in YAML files, describing inputs, outputs, and tasks.

Create a project

project top-level organizational unit to group repositories and pipelines

```
pachctl create project image-quality
```

Create repositories

repository locates data within the Pachyderm File System (PFS)

PFS version-controlled data management system backed by MinIO

```
pachctl create repo images # input data
pachctl create repo scripts # input code
# output repository created automatically
```

Add data to the repositories

Data are in backing object store (S3 or Minio), with version control and lineage tracking.

```
pachctl put file -r images@branch:/images -f /images
pachctl put file -r scripts@branch:/scripts -f /scripts
```

Create pipelines

Pipelines are a set of transformations and actions. They operate on input repositories, but also automatically create output repositories of the same name.

```
pipeline:
  name: vol
input:
  union:
    - pfs:
        repo: images
    - pfs:
        repo: scripts
  pfs:
    repo: raw_videos_and_images
    glob: "/*"
transform:
  image: ubuntu
  cmd: ["python", "/pfs/scripts/vol.py", "/pfs/images/*", "/pfs/out/"]
autoscaling: true
```

Create pipelines

Pipelines can be coarse-grained or fine-grained. In this example, they're rather fine-grained.

```
pachctl create pipeline -f vol.yaml
pachctl create pipeline -f sob.yaml
pachctl create pipeline -f fft.yaml
```

Create pipelines

There's an extra pipeline that's needed, using Pachyderm's join operator, to match filenames for vol, sob, and fft using glob patterns.

```
pipeline:
  name: vol
input:
  join:
    - pfs:
        repo: vol
        glob: "/*/*.vol"
        joinOn: "$1"
    - pfs:
        repo: sob
        glob: "/*/*.sob"
        joinOn: "$1"
    - pfs:
        repo: fft
        glob: "/*/*.fft"
        joinOn: "$1"
transform:
  image: ubuntu
  cmd: ["python", "/pfs/scripts/join.py", "/pfs/images/*"]
autoscaling: true
```

Create pipelines

It's in this joined pipeline that the `join.py` Python script (elided) calls the `cls.py` script to do the actual Naive Bayes classification of image quality.

```
pachctl create pipeline -f cls.yaml
```

What is the point of all this?

As images and scripts change in the `images` and `scripts` repositories, Pachyderm automatically versions those repositories, automatically and incrementally creates tasks to process those changes, schedules those tasks across worker pods, and versions the output, with full lineage tracking.

A picture is worth a thousand words.

NOTE: this is not my pipeline!
`https://docs.pachyderm.com/images/mldm/beginner-tutorial/dag-sixth-pipeline.webp`

Key Concepts

repositories store data (pros / cons)

pipelines process data & create more repositories

execution model you write code to read from and write to special `/pfs` filesystems

Key Concepts

scripts Bash, Python, R, Java, whatever, these typically are versioned in their own repository

images scripts run in containers which aren't exactly versioned

Pros of Pachyderm

- The Pachyderm model maps well from data pipelines built locally with scripts.
- Versioning, automatic and incremental processing, and data and code lineage are powerful.
- The execution model is also highly flexible, supporting heterogeneous pipelines.
- Model serving looks interesting, but we did not pursue this.

Cons of Pachyderm

- k8s dependency creates operational burden and a heavy lift to get started.
- The Pachyderm model is complex, with many moving parts.
- Pachyderm versioning, while adopting Git semantics, is not Git.

Alternatives to Pachyderm

dask

dask might be simpler than Pachyderm, but its execution model is different, parallelizing data structures in Python, and may be more limited. Also, it lacks versioning and automatic incremental computation.

ray

ray is also simpler than Pachyderm, and its execution model is also different. Like Pachyderm, its unit of computation is a task, but it builds on that with actors, jobs, a distributed object store, and then higher-level recipes for training, tuning, serving, etc. . . but it's all in Python.

Airflow

Airflow is well-established but is showing its age. Probably more rigid than some of the newer workflow / data processing systems.

DVC

DVC is another popular tool for DS and ML workloads, with concepts (e.g. experiments) tailored for those users. A bit like a fragmented `make` for data, it organizes dependencies among data, but leaves storage (e.g. S3) and processing (e.g. Databricks) up to you.

dbt

dbt transposes the T and the L, from ETL ELT and seems to be winning in the modern data warehouse space.

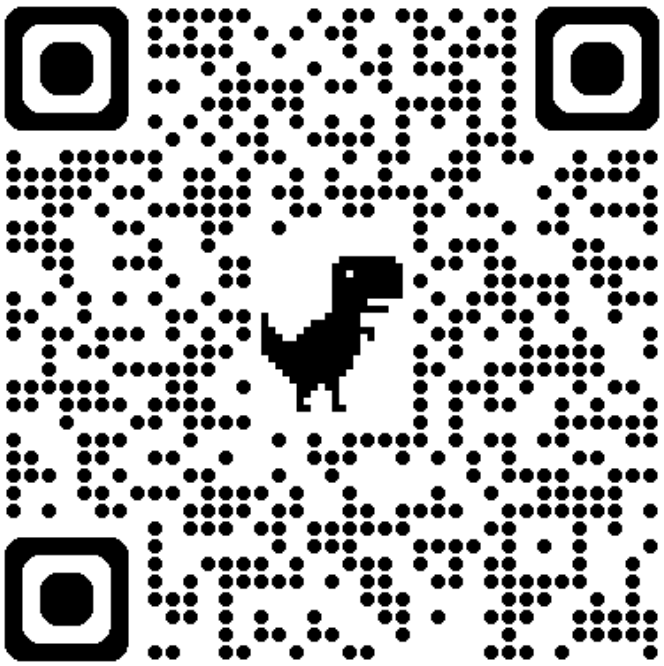
Recommendations

Assess Pachyderm *if* you have:

- k8s expertise
- complex transformation needs
- heterogeneous tooling
- more ETL than ELT
- an appetite for complexity

Resources

- Pachyderm vs Airflow (2018)
- DVC vs Airflow (2021)



https://github.com/dventimiglia/pachyderm_presentation