Pachyderm Documentation

Release 1.9.0(RC4)

Joe Doliner

Jun 16, 2019

Welcome to the Pachyderm documentation portal! Below you'll find guides and information for beginners and experienced Pachyderm users. You'll also find API references docs.

If you can't find what you're looking for or have a an issue not mentioned here, we'd love to hear from you either on GitHub, our Users Slack channel, or email us at support@pachyderm.io.

Note: if you are using a Pachyderm version < 1.4, you can find relevant docs here.

Getting Started

Welcome to the documentation portal for first time Pachyderm users! We've organized information into two sections:

1. Local Installation: Get Pachyderm deployed locally on macOS or Linux.
2. Beginner Tutorial: Learn to use Pachdyerm through a quick and simple tutorial.

If you'd like to read about the Pachyderm's open source and enterprise features before actually running it, check out the following:

- The open source core
- Enterprise edition
- Use Cases

Looking for more in-depth development docs? Check out the Pachyderm fundamentals:

- Getting data into Pachyderm
- Creating analysis pipelines
- Distributed computing
- Incremental processing
- Getting data out of Pachyderm
- Updating pipelines

Need to see different or more advanced Pachyderm examples? You can find a bunch of them here.

Note - If you've already got a Kubernetes cluster running or would rather use AWS, GCE or Azure to deploy Pachyderm, check out our deployment guides.

CHAPTER 2

Local Installation

This guide will walk you through the recommended path to get Pachyderm running locally on macOS or Linux.

If you hit any errors not covered in this guide, check our general troubleshooting docs for common errors, submit an issue on GitHub, join our users channel on Slack, or email us at support@pachyderm.io and we can help you right away.

2.1 Prerequisites

- *Minikube* (and VirtualBox) or *Docker Desktop (v18.06+)*
- *Pachyderm Command Line Interface*

2.1.1 Minikube

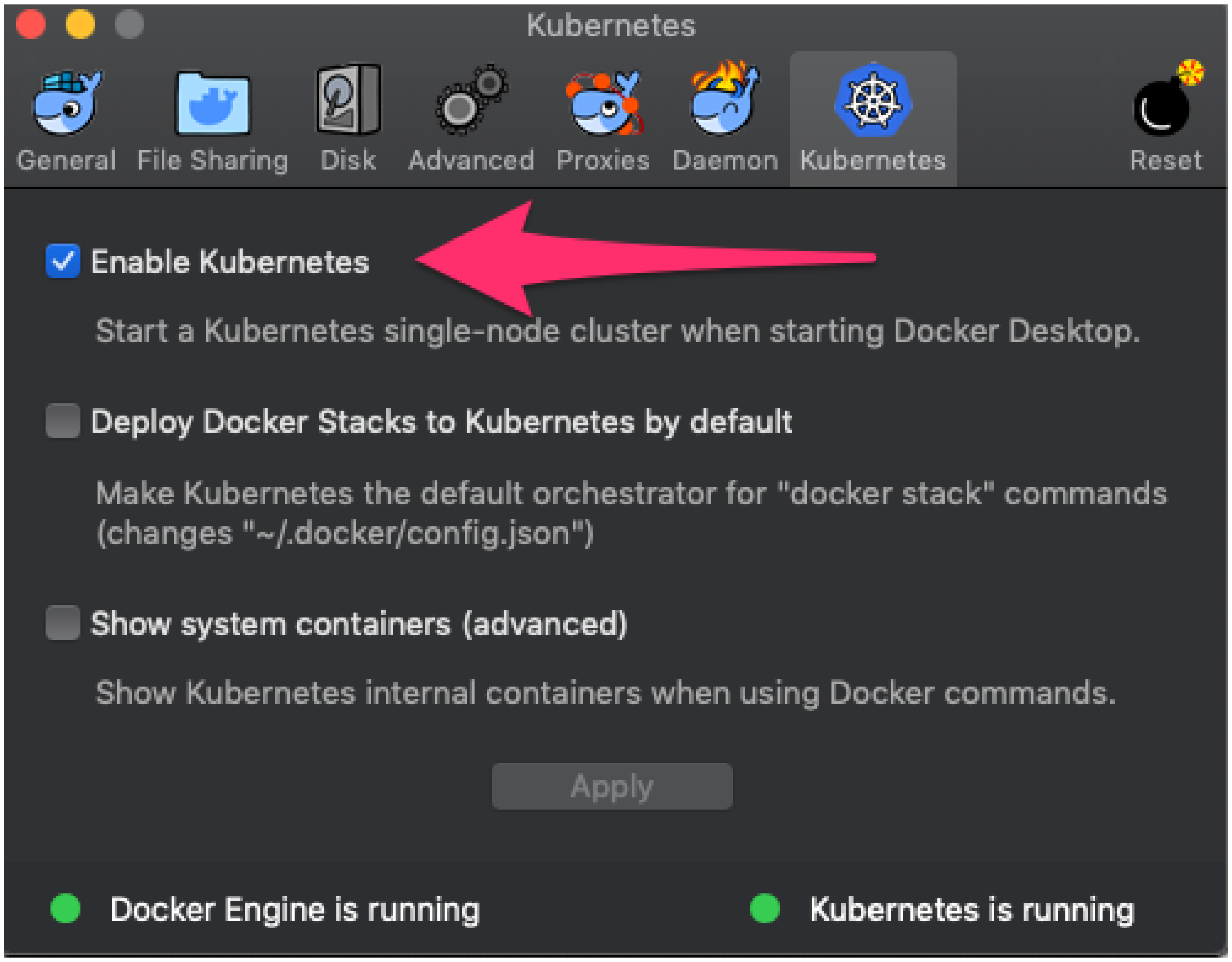
Kubernetes offers an excellent guide to install minikube. Follow the Kubernetes installation guide to install Virtual Box, Minikube, and Kubectl. Then come back here to start Minikube:

```
minikube start
```

Note: Any time you want to stop and restart Pachyderm, you should start fresh with `minikube delete` and `minikube start`. Minikube isn't meant to be a production environment and doesn't handle being restarted well without a full wipe.

2.1.2 Docker Desktop

First you need to make sure kubernetes is enabled in the docker desktop settings



✓ Enable Kubernetes

Start a Kubernetes single-node cluster when starting Docker Desktop.

Deploy Docker Stacks to Kubernetes by default

Make Kubernetes the default orchestrator for "docker stack" commands (changes "~/.docker/config.json")

Show system containers (advanced)

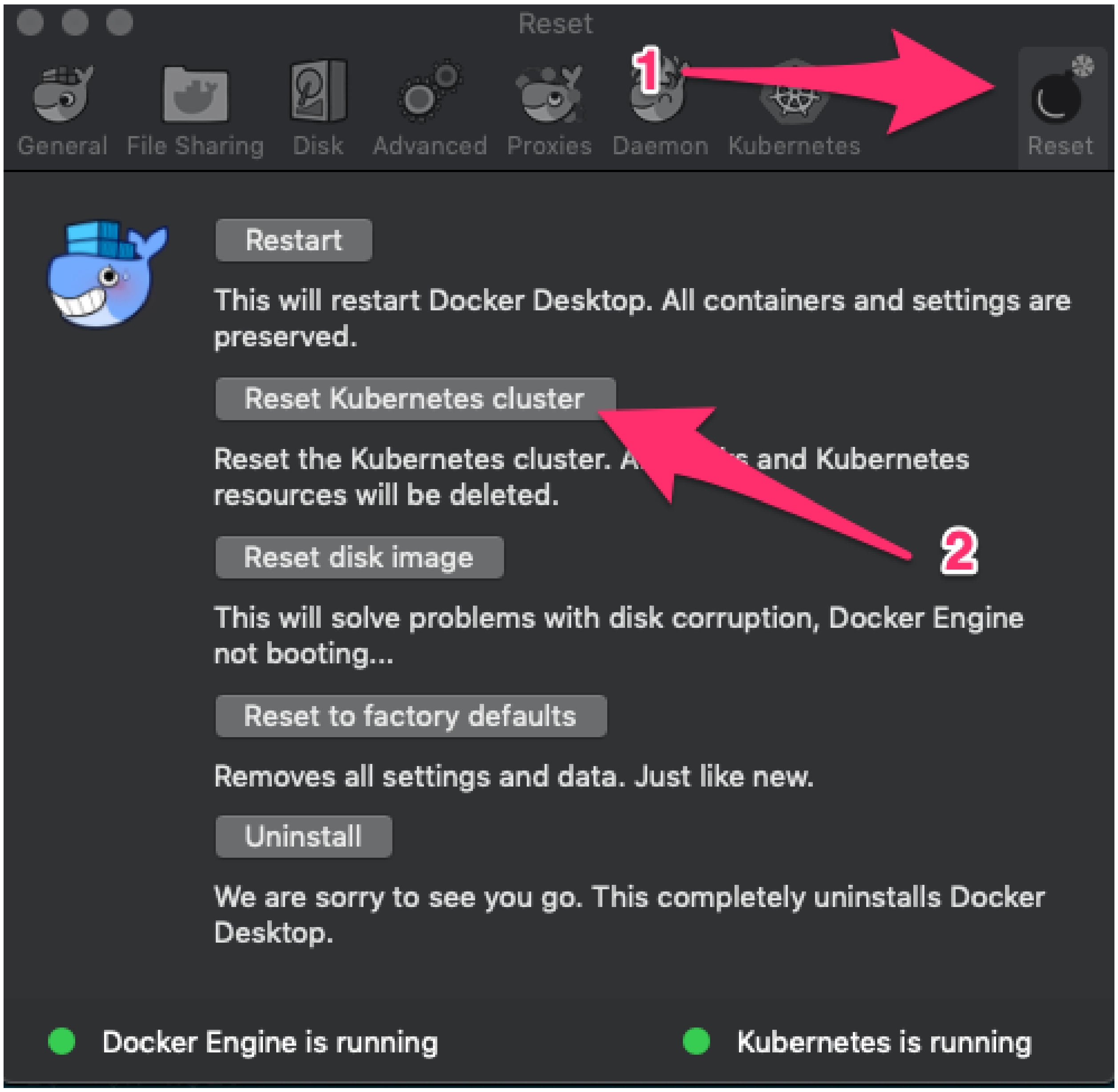
Show Kubernetes internal containers when using Docker commands.

● Docker Engine is running ● Kubernetes is running

And then confirm things are running

```
$ kubectl get all
NAME                  TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)   AGE
service/kubernetes    ClusterIP   10.96.0.1     <none>        443/TCP   56d
```

To reset your kubernetes cluster on Docker For Desktop just click the reset button in the preferences section



Restart

This will restart Docker Desktop. All containers and settings are preserved.

Reset Kubernetes cluster

Reset the Kubernetes cluster. A[illegible] and Kubernetes resources will be deleted.

Reset disk image

This will solve problems with disk corruption, Docker Engine not booting...

Reset to factory defaults

Removes all settings and data. Just like new.

Uninstall

We are sorry to see you go. This completely uninstalls Docker Desktop.

Docker Engine is running

Kubernetes is running

2.1.3 Pachctl

pachctl is a command-line utility used for interacting with a Pachyderm cluster.

```
# For macOS:
$ brew tap pachyderm/tap && brew install pachyderm/tap/pachctl@1.9

# For Debian based linux (64 bit) or Window 10+ on WSL:
$ curl -o /tmp/pachctl.deb -L https://github.com/pachyderm/pachyderm/releases/
↪download/v1.9.0/pachctl_1.9.0_amd64.deb && sudo dpkg -i /tmp/pachctl.deb

# For all other linux flavors
$ curl -o /tmp/pachctl.tar.gz -L https://github.com/pachyderm/pachyderm/releases/
↪download/v1.9.0/pachctl_1.9.0_linux_amd64.tar.gz && tar -xvf /tmp/pachctl.tar.gz -C␣
↪/tmp && sudo cp /tmp/pachctl_1.9.0_linux_amd64/pachctl /usr/local/bin
```

Note: To install an older version of Pachyderm, navigate to that version using the menu in the bottom left.

To check that installation was successful, you can try running `pachctl help`, which should return a list of Pachyderm commands.

2.2 Deploy Pachyderm

Now that you have Minikube running, it's incredibly easy to deploy Pachyderm.

```
$ pachctl deploy local
```

This generates a Pachyderm manifest and deploys Pachyderm on Kubernetes. It may take a few minutes for the Pachyderm pods to be in a `Running` state, because the containers have to be pulled from DockerHub. You can see the status of the Pachyderm pods using `kubectl get pods`. When Pachyderm is ready for use, this should return something similar to:

```
$ kubectl get pods
NAME                         READY     STATUS    RESTARTS   AGE
dash-6c9dc97d9c-vb972        2/2       Running   0          6m
etcd-7dbb489f44-9v5jj        1/1       Running   0          6m
pachd-6c878bbc4c-f2h2c       1/1       Running   0          6m
```

Note: If you see a few restarts on the `pachd` nodes, that's ok. That simply means that Kubernetes tried to bring up those pods before `etcd` was ready so it restarted them.

Try `pachctl version` to make sure everything is working.

```
$ pachctl version
COMPONENT           VERSION
pachctl             1.8.2
pachd               1.8.2
```

We're good to go!

`pachctl` uses port forwarding by default. This is slower than if you connect directly by the minikube instance, like so:

```
# Find the IP address
$ minikube ip
192.168.99.100

# Set the `PACHD_ADDRESS` environment variable
$ export PACHD_ADDRESS=192.168.99.100:30650

# Run a command
$ pachctl version
```

Note: `ADDRESS` was renamed to `PACHD_ADDRESS` in 1.8.3. If you are using an older version of Pachyderm, use the `ADDRESS` environment variable instead.

2.3 Next Steps

Now that you have everything installed and working, check out our Beginner Tutorial to learn the basics of Pachyderm such as adding data and building pipelines for analysis.

The Pachyderm Enterprise dashboard is deployed by default with Pachyderm. We offer a FREE trial token to experiment with this interface to Pachyderm. To check it out, first enable port forwarding via `pachctl port-forward`, then point your web browser to `localhost:30080`. Alternatively, if you set the `PACHD_ADDRESS` environment variable like in the previous section, you can circumvent port forwarding by just pointing your web browser to port 30080 on your minikube IP address.

CHAPTER 3

Beginner Tutorial

Welcome to the beginner tutorial for Pachyderm! If you've already got Pachyderm installed, this guide should take about 15 minutes, and it will introduce you to the basic concepts of Pachyderm.

3.1 Image processing with OpenCV

This guide will walk you through the deployment of a Pachyderm pipeline to do some simple edge detection on a few images. Thanks to Pachyderm's built-in processing primitives, we'll be able to keep our code simple but still run the pipeline in a distributed, streaming fashion. Moreover, as new data is added, the pipeline will automatically process it and output the results.

If you hit any errors not covered in this guide, get help in our public community Slack, submit an issue on GitHub, or email us at support@pachyderm.io. We are more than happy to help!

3.1.1 Prerequisites

This guide assumes that you already have Pachyderm running locally. Check out our *Local Installation* instructions if haven't done that yet and then come back here to continue.

3.1.2 Create a Repo

A `repo` is the highest level data primitive in Pachyderm. Like many things in Pachyderm, it shares its name with a primitive in Git and is designed to behave analogously. Generally, repos should be dedicated to a single source of data such as log messages from a particular service, a users table, or training data for an ML model. Repos are dirt cheap so don't be shy about making tons of them.

For this demo, we'll simply create a repo called `images` to hold the data we want to process:

```
$ pachctl create repo images
$ pachctl list repo
NAME   CREATED       SIZE (MASTER)
images 7 seconds ago 0B
```

This shows that the repo has been successfully created, and the size of repo's HEAD commit on the master branch is 0B, since we haven't added anything to it yet.

3.1.3 Adding Data to Pachyderm

Now that we've created a repo it's time to add some data. In Pachyderm, you write data to an explicit `commit` (again, similar to Git). Commits are immutable snapshots of your data which give Pachyderm its version control properties. `Files` can be added, removed, or updated in a given commit.

Let's start by just adding a file, in this case an image, to a new commit. We've provided some sample images for you that we host on Imgur.

We'll use the `put file` command along with the `-f` flag. `-f` can take either a local file, a URL, or a object storage bucket which it'll automatically scrape. In our case, we'll simply pass the URL.

Unlike Git, commits in Pachyderm must be explicitly started and finished as they can contain huge amounts of data and we don't want that much "dirty" data hanging around in an unpersisted state. `put file` automatically starts and finishes a commit for you so you can add files more easily. If you want to add many files over a period of time, you can do `start commit` and `finish commit` yourself.

We also specify the repo name "images", the branch name "master", and the file name: "liberty.png".

Here's an example atomic commit of the file `liberty.png` to the `images` repo's `master` branch:

```
$ pachctl put file images@master:liberty.png -f http://imgur.com/46Q8nDz.png
```

We can check to make sure the data we just added is in Pachyderm.

```
# If we list the repos, we can see that there is now data
$ pachctl list repo
NAME   CREATED            SIZE (MASTER)
images About a minute ago 57.27KiB

# We can view the commit we just created
$ pachctl list commit images
REPO   COMMIT                           PARENT STARTED        DURATION           SIZE
images d89758a7496a4c56920b0eaa7d7d3255 <none> 29 seconds ago Less than a second 57.
↪27KiB

# And view the file in that commit
$ pachctl list file images@master
COMMIT                           NAME         TYPE COMMITTED          SIZE
d89758a7496a4c56920b0eaa7d7d3255 /liberty.png file About a minute ago 57.27KiB
```

We can also view the file we just added to Pachyderm. Since this is an image, we can't just print it out in the terminal, but the following commands will let you view it easily.

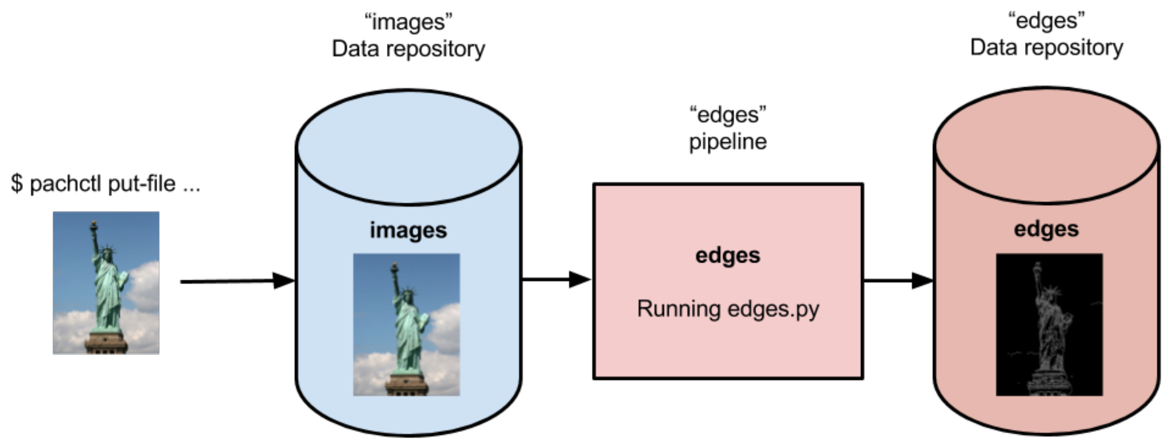
```
# on macOS
$ pachctl get file images@master:liberty.png | open -f -a /Applications/Preview.app

# on Linux
$ pachctl get file images@master:liberty.png | display
```

3.1.4 Create a Pipeline

Now that we've got some data in our repo, it's time to do something with it. `Pipelines` are the core processing primitive in Pachyderm and they're specified with a JSON encoding. For this example, we've already created the pipeline for you and you can find the code on Github.

When you want to create your own pipelines later, you can refer to the full *Pipeline Specification* to use more advanced options. Options include building your own code into a container instead of the pre-built Docker image we'll be using here.

For now, we're going to create a single pipeline that takes in images and does some simple edge detection.



Below is the pipeline spec and python code we're using. Let's walk through the details.

```
# edges.json
{
  "pipeline": {
    "name": "edges"
  },
  "transform": {
    "cmd": [ "python3", "/edges.py" ],
    "image": "pachyderm/opencv"
  },
  "input": {
    "pfs": {
      "repo": "images",
      "glob": "/*"
    }
  }
}
```

Our pipeline spec contains a few simple sections. First is the pipeline `name` , edges. Then we have the `transform` which specifies the docker image we want to use, `pachyderm/opencv` (defaults to DockerHub as the registry), and the entry point `edges.py` . Lastly, we specify the input. Here we only have one PFS input, our images repo with a particular glob pattern.

The glob pattern defines how the input data can be broken up if we want to distribute our computation. `/*` means that each file can be processed individually, which makes sense for images. Glob patterns are one of the most powerful features of Pachyderm so when you start creating your own pipelines, check out the *Pipeline Specification*.

```
# edges.py
import cv2
import numpy as np
from matplotlib import pyplot as plt
import os

# make_edges reads an image from /pfs/images and outputs the result of running
# edge detection on that image to /pfs/out. Note that /pfs/images and
# /pfs/out are special directories that Pachyderm injects into the container.
def make_edges(image):
   img = cv2.imread(image)
   tail = os.path.split(image)[1]
   edges = cv2.Canny(img,100,200)
   plt.imsave(os.path.join("/pfs/out", os.path.splitext(tail)[0]+'.png'), edges, cmap␣
↪= 'gray')

# walk /pfs/images and call make_edges on every file found
for dirpath, dirs, files in os.walk("/pfs/images"):
   for file in files:
       make_edges(os.path.join(dirpath, file))
```

We simply walk over all the images in `/pfs/images`, do our edge detection, and write to `/pfs/out`.

`/pfs/images` and `/pfs/out` are special local directories that Pachyderm creates within the container automatically. All the input data for a pipeline will be found in `/pfs/<input_repo_name>` and your code should always write out to `/pfs/out`. Pachyderm will automatically gather everything you write to `/pfs/out` and version it as this pipeline's output.

Now let's create the pipeline in Pachyderm:

```
$ pachctl create pipeline -f https://raw.githubusercontent.com/pachyderm/pachyderm/
↪master/examples/opencv/edges.json
```

3.1.5 What Happens When You Create a Pipeline

Creating a pipeline tells Pachyderm to run your code on the data in your input repo (the HEAD commit) as well as **all future commits** that occur after the pipeline is created. Our repo already had a commit, so Pachyderm automatically launched a `job` to process that data.

The first time Pachyderm runs a pipeline job, it needs to download the Docker image (specified in the pipeline spec) from the specified Docker registry (DockerHub in this case). This first run this might take a minute or so because of the image download, depending on your Internet connection. Subsequent runs will be much faster.

You can view the job with:

```
$ pachctl list job
ID                                 PIPELINE STARTED        DURATION          RESTART␣
↪PROGRESS  DL       UL       STATE
0f6a53829eeb4ca193bb7944fe693700 edges    16 seconds ago Less than a second 0       1␣
↪+ 0 / 1 57.27KiB 22.22KiB success
```

Yay! Our pipeline succeeded! Pachyderm creates a corresponding output repo for every pipeline. This output repo will have the same name as the pipeline, and all the results of that pipeline will be versioned in this output repo. In our example, the "edges" pipeline created a repo called "edges" to store the results.

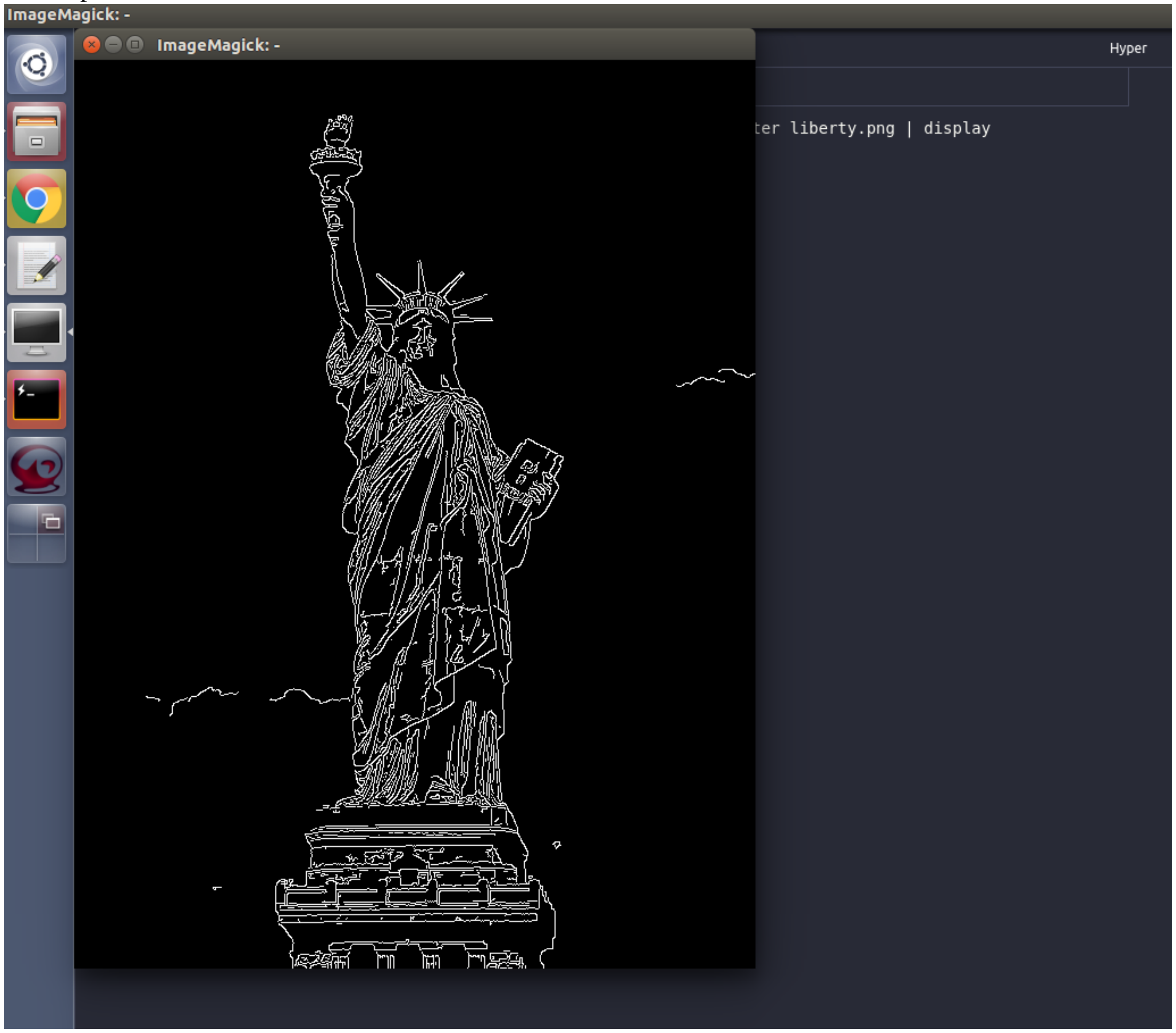
```
$ pachctl list repo
NAME   CREATED        SIZE (MASTER)
edges  2 minutes ago 22.22KiB
images 5 minutes ago 57.27KiB
```

3.1.6 Reading the Output

We can view the output data from the "edges" repo in the same fashion that we viewed the input data.

```
# on macOS
$ pachctl get file edges@master:liberty.png | open -f -a /Applications/Preview.app

# on Linux
$ pachctl get file edges@master:liberty.png | display
```

The output should look similar to:



```
ter liberty.png | display
```

3.1.7 Processing More Data

Pipelines will also automatically process the data from new commits as they are created. Think of pipelines as being subscribed to any new commits on their input repo(s). Also similar to Git, commits have a parental structure that tracks which files have changed. In this case we're going to be adding more images.

Let's create two new commits in a parental structure. To do this we will simply do two more `put file` commands and by specifying `master` as the branch, it'll automatically parent our commits onto each other. Branch names are just references to a particular HEAD commit.

```
$ pachctl put file images@master:AT-AT.png -f http://imgur.com/8MN9Kg0.png

$ pachctl put file images@master:kitten.png -f http://imgur.com/g2QnNqa.png
```

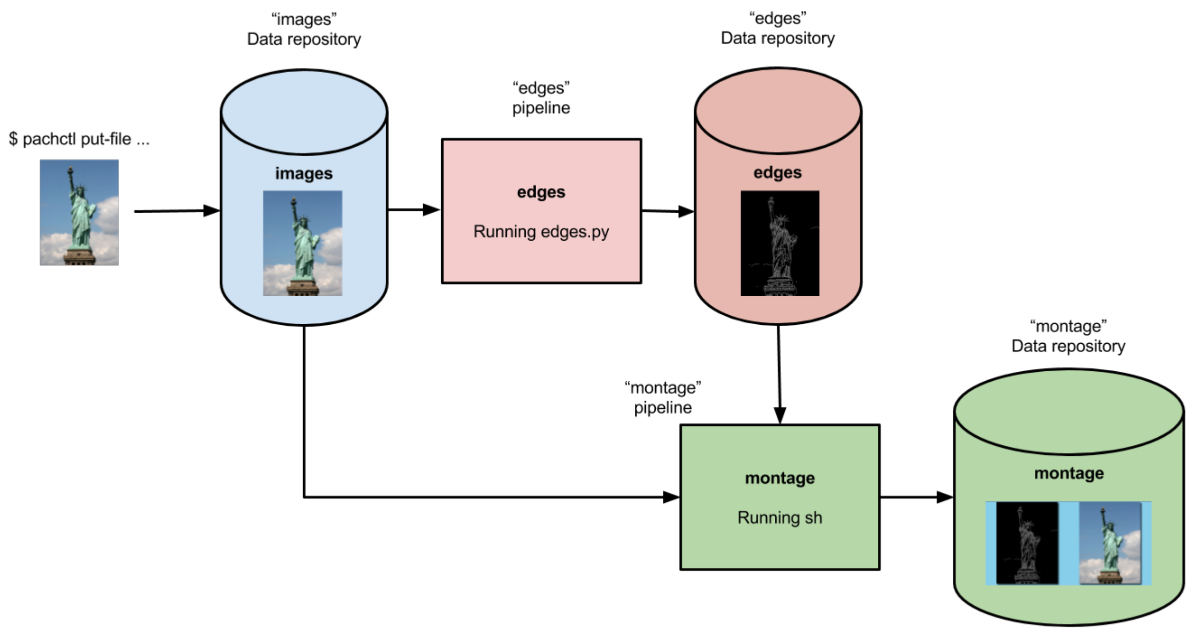
Adding a new commit of data will automatically trigger the pipeline to run on the new data we've added. We'll see corresponding jobs get started and commits to the output "edges" repo. Let's also view our new outputs.

```
# view the jobs that were kicked off
$ pachctl list job
ID                                STARTED        DURATION           RESTART PROGRESS  ␣
↪DL        UL        STATE
81ae47a802f14038b95f8f248cddbed2  7 seconds ago  Less than a second 0       1 + 2 / 3␣
↪102.4KiB 74.21KiB success
ce448c12d0dd4410b3a5ae0c0f07e1f9  16 seconds ago Less than a second 0       1 + 1 / 2␣
↪78.7KiB  37.15KiB success
490a28be32de491e942372018cd42460  9 minutes ago  35 seconds         0       1 + 0 / 1␣
↪57.27KiB 22.22KiB success
```

```
# View the output data

# on macOS
$ pachctl get file edges@master:AT-AT.png | open -f -a /Applications/Preview.app

$ pachctl get file edges@master:kitten.png | open -f -a /Applications/Preview.app

# on Linux
$ pachctl get file edges@master:AT-AT.png | display

$ pachctl get file edges@master:kitten.png | display
```

3.1.8 Adding Another Pipeline

We have succesfully deployed and used a single stage Pachyderm pipeline. Now let's add a processing stage to illustrate a multi-stage Pachyderm pipeline. Specifically, let's add a `montage` pipeline that take our original and edge detected images and arranges them into a single montage of images:



Below is the pipeline spec for this new pipeline:

```
# montage.json
{
  "pipeline": {
    "name": "montage"
  },
  "input": {
    "cross": [ {
      "pfs": {
        "glob": "/",
        "repo": "images"
      }
    },
    {
      "pfs": {
        "glob": "/",
        "repo": "edges"
      }
    } ]
  },
  "transform": {
    "cmd": [ "sh" ],
    "image": "v4tech/imagemagick",
    "stdin": [ "montage -shadow -background SkyBlue -geometry 300x300+2+2 $(find /pfs␣
↪-type f | sort) /pfs/out/montage.png" ]
  }
}
```

This `montage` pipeline spec is similar to our `edges` pipeline except for three differences: (1) we are using a different Docker image that has imagemagick installed, (2) we are executing a `sh` command with `stdin` instead of a python script, and (3) we have multiple input data repositories.

In the `montage` pipeline we are combining our multiple input data repositories using a `cross` pattern. This `cross` pattern creates a single pairing of our input images with our edge detected images. There are several interesting ways to combine data in Pachyderm, which are discussed here and here.

We create the `montage` pipeline as before, with `pachctl`:

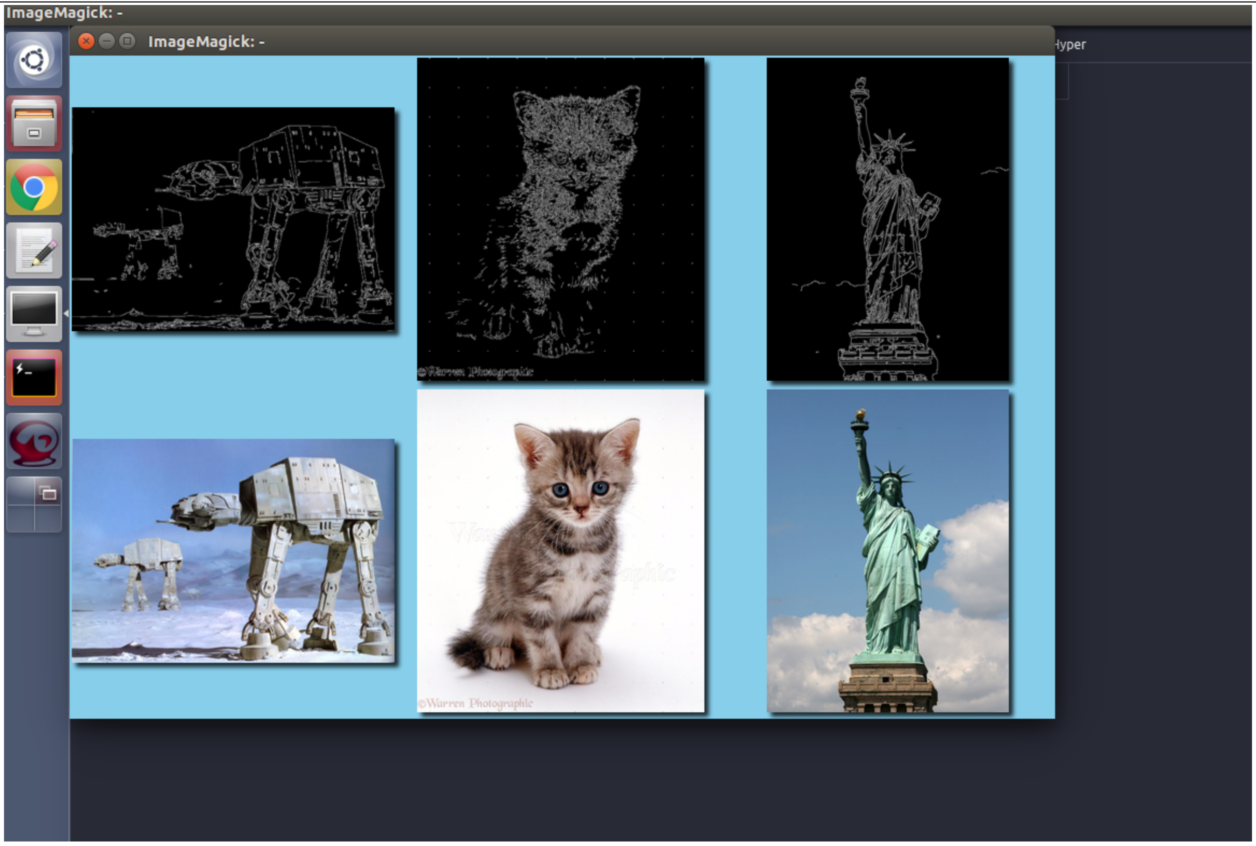
```
$ pachctl create pipeline -f https://raw.githubusercontent.com/pachyderm/pachyderm/
↪master/examples/opencv/montage.json
```

Pipeline creating triggers a job that generates a montage for all the current HEAD commits of the input repos:

```
$ pachctl list job
ID                                  STARTED        DURATION           RESTART␣
↪PROGRESS  DL        UL        STATE
92cecc40c3144fd5b4e07603bb24b104    45 seconds ago 6 seconds          0        1 + 0 /␣
↪1 371.9KiB 1.284MiB success
81ae47a802f14038b95f8f248cddbed2    2 minutes ago  Less than a second 0        1 + 2 /␣
↪3 102.4KiB 74.21KiB success
ce448c12d0dd4410b3a5ae0c0f07e1f9    2 minutes ago  Less than a second 0        1 + 1 /␣
↪2 78.7KiB  37.15KiB success
490a28be32de491e942372018cd42460    11 minutes ago 35 seconds         0        1 + 0 /␣
↪1 57.27KiB 22.22KiB success
```

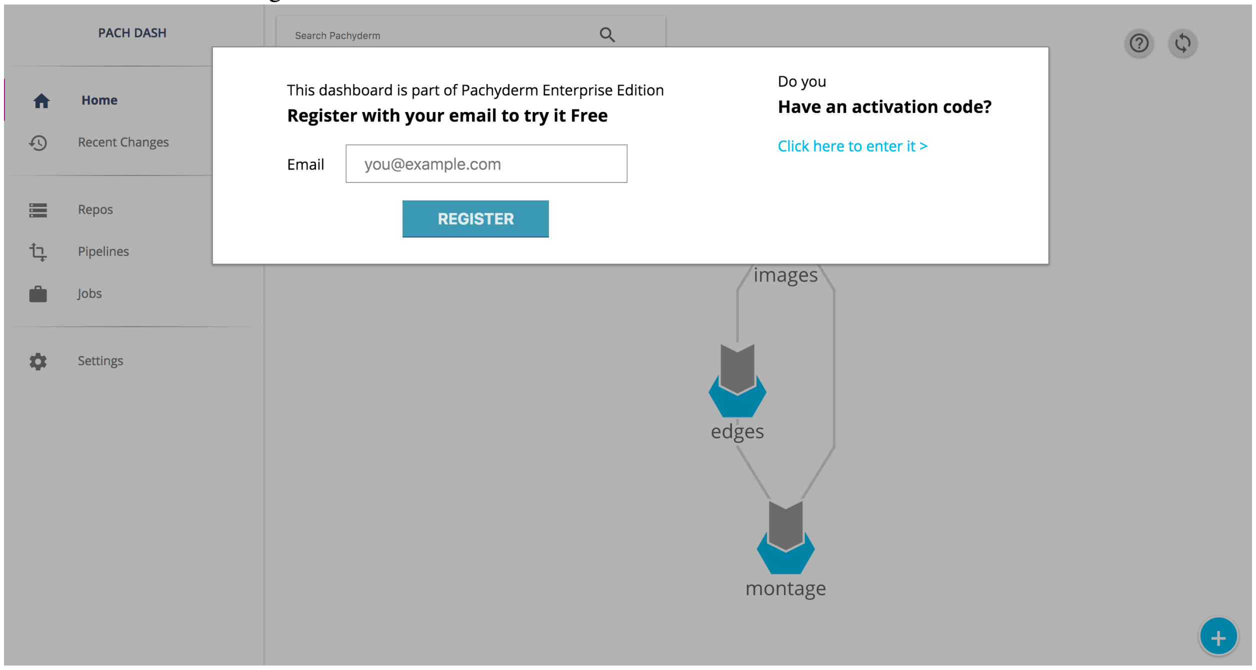
And you can view the generated montage image via:

```
# on macOS
$ pachctl get file montage@master:montage.png | open -f -a /Applications/Preview.app

# on Linux
$ pachctl get file montage@master:montage.png | display
```



3.2 Exploring your DAG in the Pachyderm dashboard

When you deployed Pachyderm locally, the Pachyderm Enterprise dashboard was also deployed by default. This dashboard will let you interactively explore your pipeline, visualize the structure of the pipeline, explore your data, debug jobs, etc. To access the dashboard visit `localhost:30080` in an Internet browser (e.g., Google Chrome). You should see something similar to this:



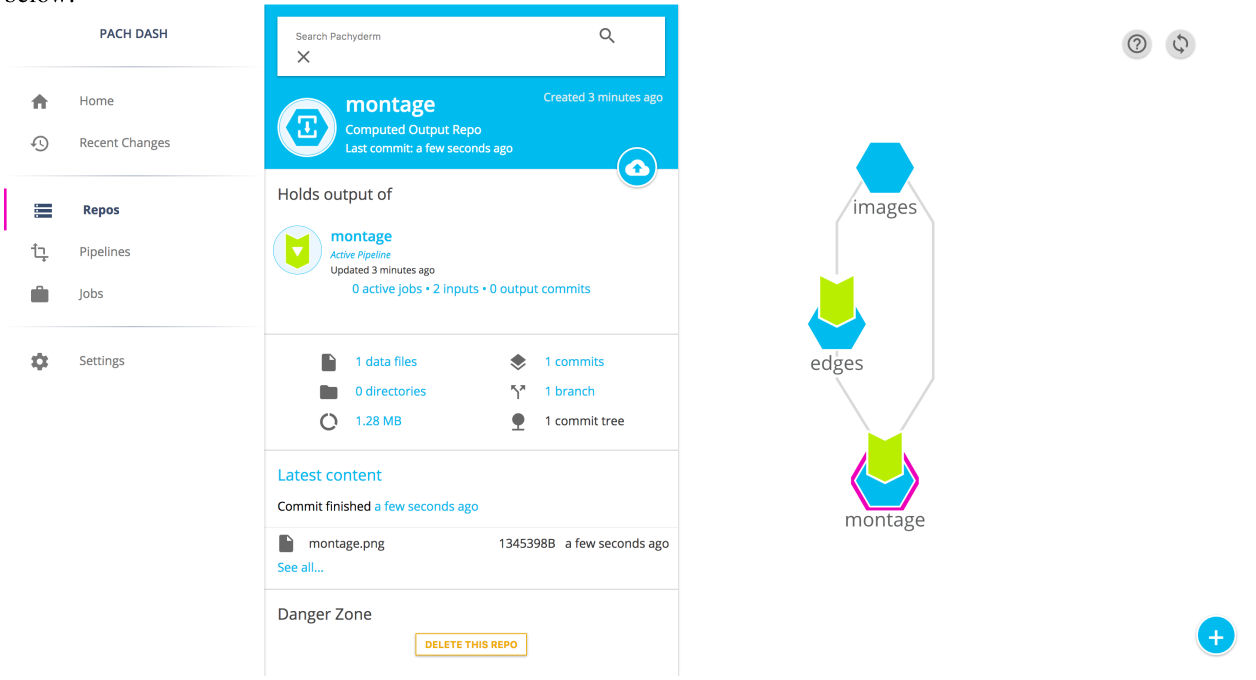
This dashboard is part of Pachyderm Enterprise Edition
Register with your email to try it Free

Email: you@example.com

REGISTER

Do you
Have an activation code?
Click here to enter it >

Enter your email address if you would like to obtain a free trial token for the dashboard. Upon entering this trial token, you will be able to see your pipeline structure and interactively explore the various pieces of your pipeline as pictured below:



montage
Computed Output Repo
Last commit: a few seconds ago
Created 3 minutes ago

Holds output of

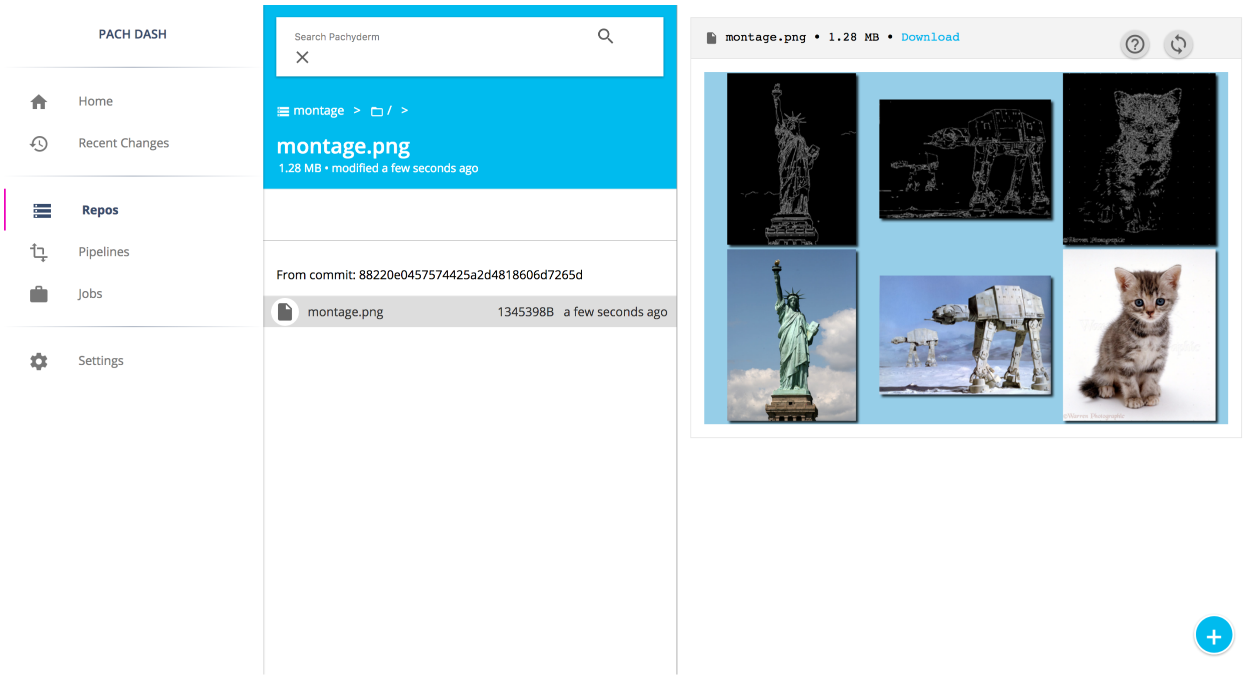
montage
Active Pipeline
Updated 3 minutes ago
0 active jobs • 2 inputs • 0 output commits

1 data files
1 commits
0 directories
1 branch
1.28 MB
1 commit tree

Latest content
Commit finished a few seconds ago

montage.png 1345398B a few seconds ago
See all...

Danger Zone
DELETE THIS REPO



From commit: 88220e0457574425a2d4818606d7265d

montage.png 1345398B a few seconds ago

3.3 Next Steps

Pachyderm is now running locally with data and a pipeline! To play with Pachyderm locally, you can use what you've learned to build on or change this pipeline. You can also dig in and learn more details about:

- Deploying Pachyderm to the cloud or on prem
- *Getting Your Data into Pachyderm*
- *Creating Analysis Pipelines*

We'd love to help and see what you come up with, so submit any issues/questions you come across on GitHub , Slack, or email at support@pachyderm.io if you want to show off anything nifty you've created!

CHAPTER 4

Getting Your Data into Pachyderm

Data that you put (or "commit") into Pachyderm ultimately lives in an object store of your choice (S3, Minio, GCS, etc.). This data is content-addressed by Pachyderm to build our version control semantics for data and is therefore not "human-readable" directly in the object store. That being said, Pachyderm allows you and your pipeline stages to interact with versioned data like you would in a normal file system.

4.1 Jargon associated with putting data in Pachyderm

4.1.1 "Data Repositories"

Versioned data in Pachyderm lives in repositories (again think about something similar to "git for data"). Each data "repository" can contain one file, multiple files, multiple files arranged in directories, etc. Regardless of the structure, Pachyderm will version the state of each data repository as it changes over time.

4.1.2 "Commits"

Regardless of the method you use to get data into Pachyderm (CLI, language client, etc.), the mechanism that is used is a "commit" of data into a data repository. In order to put data into Pachyderm, a commit must be "started" (aka an "open commit"). Data can then be put into Pachyderm as part of that open commit and will be available once the commit is "finished" (aka a "closed commit").

4.2 How to get data into Pachyderm

In terms of actually getting data into Pachyderm via "commits," there are a few options:

- *Via the `pachctl` CLI tool*: This is the great option for testing, development, integration with CI/CD, and for users who prefer scripting.
- *Via one of the Pachyderm language clients*: This option is ideal for Go, Python, or Scala users who want to push data to Pachyderm from services or applications written in those languages. Actually, even if you don't use Go, Python, or Scala, Pachyderm uses a protobuf API which supports many other languages, we just haven't built the full clients yet.

If you're on Pachyderm Enterprise, you additionally get these options:

- *Via the s3gateway*: This is ideal when using existing tools or libraries that interact with object stores.

- *Via the Pachyderm dashboard*: The Pachyderm Enterprise dashboard provides a very convenient way to upload data right from the GUI. You can find out more about Pachyderm Enterprise Edition here.

4.2.1 pachctl

To get data into Pachyderm using `pachctl`, you first need to create one or more data repositories to hold your data:

```
$ pachctl create repo <repo name>
```

Then, to put data into the created repo, you use the `put file` command. Below are a few example uses of `put file`, but you can see the complete documentation here.

If there is an open commit, `put file` will add files to that commit. This example will add two files to a new commit, then close the commit:

```
# first start a commit
$ pachctl start commit <repo>@<branch>

# put <file1> in the <repo> on <branch>
$ pachctl put file <repo>@<branch>:</path/to/file1> -f <file1>

# put <file2> in the <repo> on <branch>
$ pachctl put file <repo>@<branch>:</path/to/file2> -f <file2>

# then finish the commit
$ pachctl finish commit <repo>@<branch>
```

If there is not an open commit, `put file` will implicitly start and finish the commit. This is called an atomic commit:

```
$ pachctl put file <repo>@<branch>:</path/to/file> -f <file>
```

Put data from a URL:

```
$ pachctl put file <repo>@<branch>:</path/to/file> -f http://url_path
```

Put data directly from an object store:

```
# here you can use s3://, gcs://, or as://
$ pachctl put file <repo>@<branch>:</path/to/file> -f s3://object_store_url
```

Add multiple files at once by using the `-i` option or multiple `-f` flags. In the case of `-i`, the target file should be a list of files, paths, or URLs that you want to input all at once:

```
$ pachctl put file <repo>@<branch> -i <file containing list of files, paths, or URLs>
```

Pipe data from stdin into a data repository:

```
$ echo "data" | pachctl put file <repo>@<branch> -f </path/to/file>
```

Add an entire directory or all of the contents at a particular URL (either HTTP(S) or object store URL, `s3://`, `gcs://`, and `as://`) by using the recursive flag, `-r`:

```
$ pachctl put file <repo>@<branch> -r -f <dir>
```

4.2.2 Pachyderm Language Clients

There are a number of Pachyderm language clients. These can be used to programmatically put data into Pachyderm, and much more. You can find out more about these clients here.

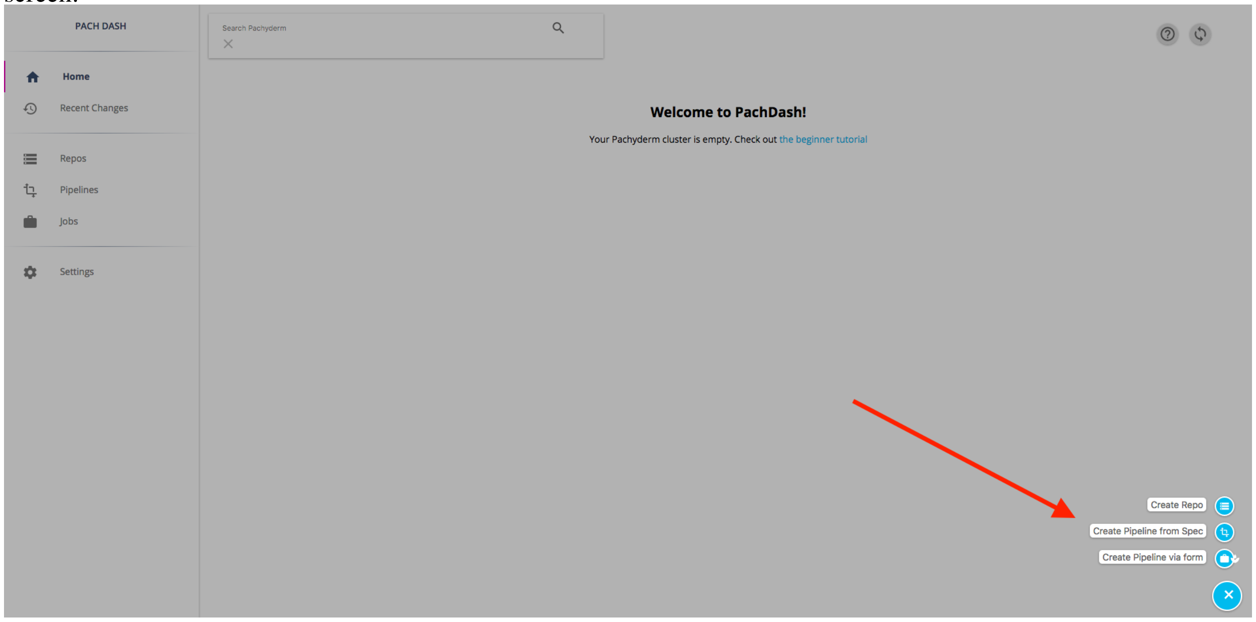
4.2.3 The S3Gateway

We support an HTTP API that offers a subset of S3's functionality. With this, you can use existing tools or libraries that work with object stores, such as minio. See the s3gateway docs for more information.

4.2.4 The Pachyderm Dashboard

When you deployed Pachyderm, the Pachyderm Enterprise dashboard was also deployed automatically (if you followed one of our deploy guides here). You can get a FREE trial token to experiment with this dashboard, which will let you create data repositories and add data to those repositories via a GUI. More information about getting your FREE trial token and activating the dashboard can be found here.

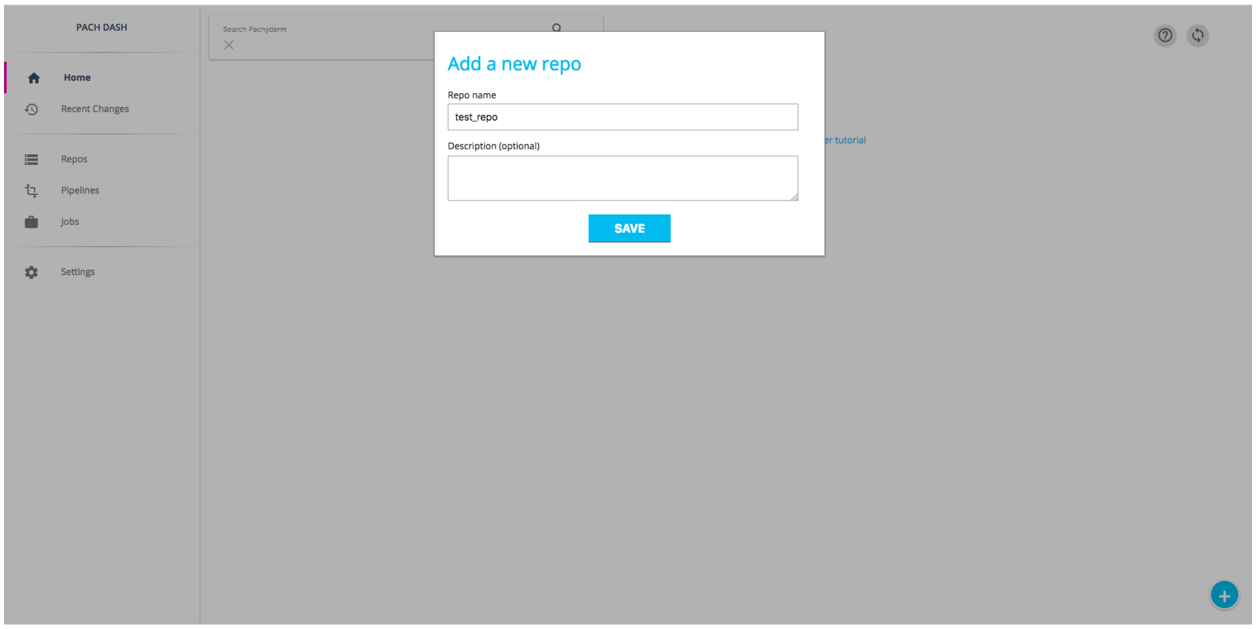
In the dashboard, you can create a data repository by clicking on the + sign icon in the lower right hand corner of the screen:



Welcome to PachDash!

Your Pachyderm cluster is empty. Check out the beginner tutorial

When you click "Create Repo," a box will pop up prompting you for a name and optional description for the repo:



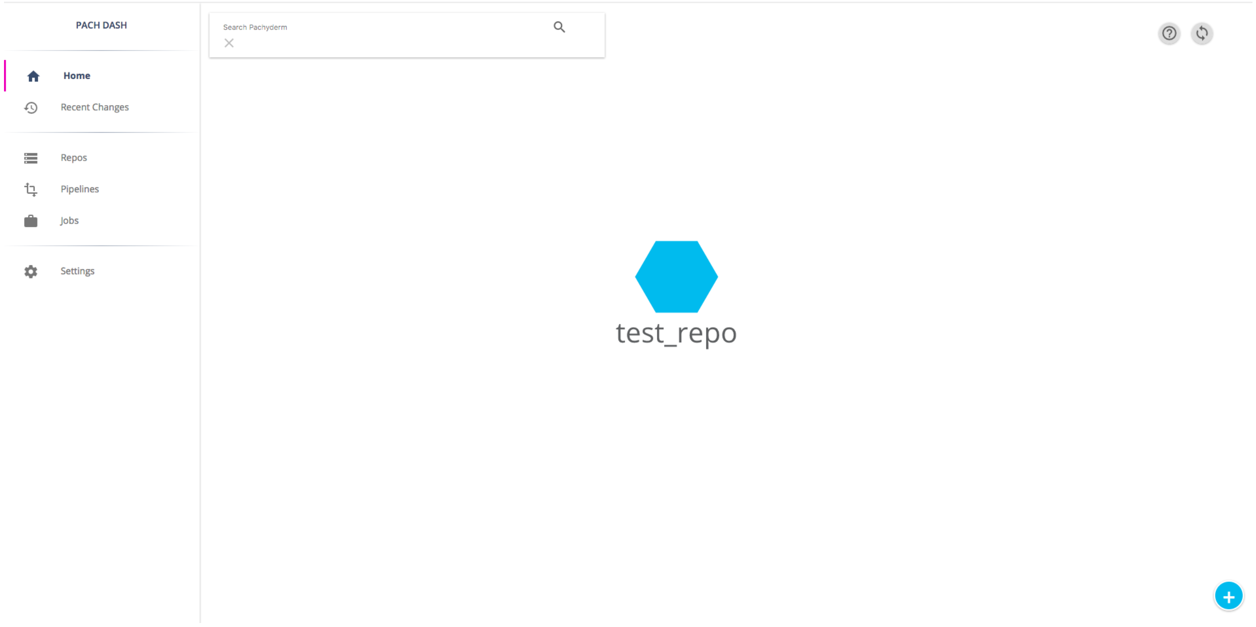
Add a new repo

Repo name: test_repo

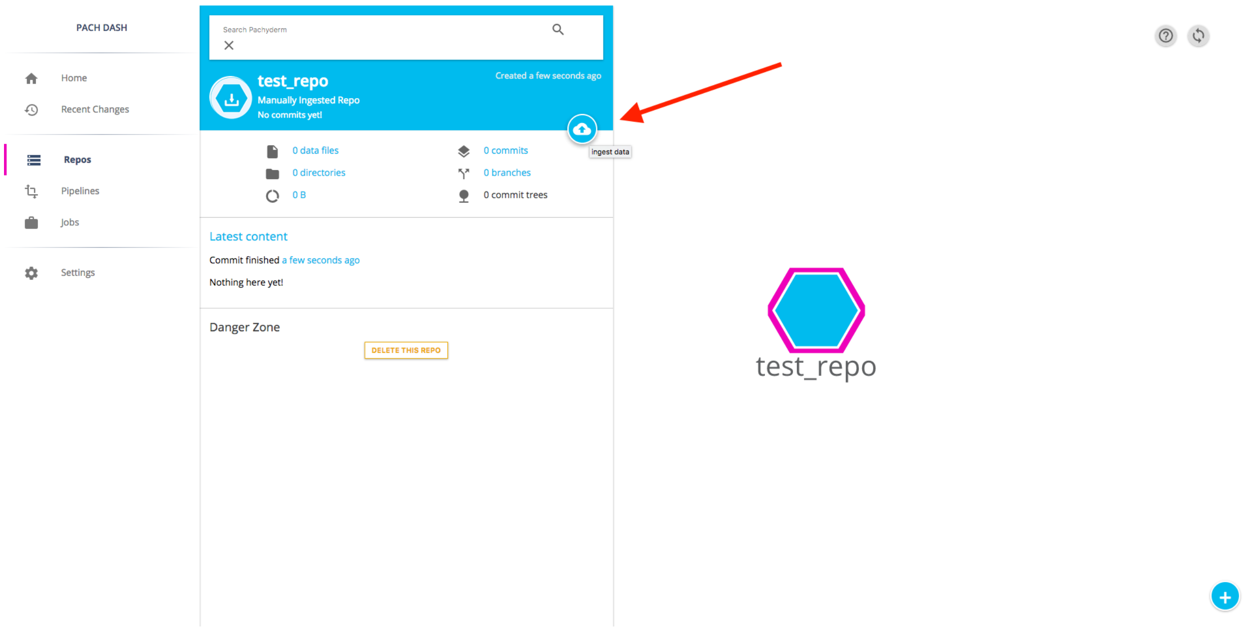
Description (optional):

SAVE

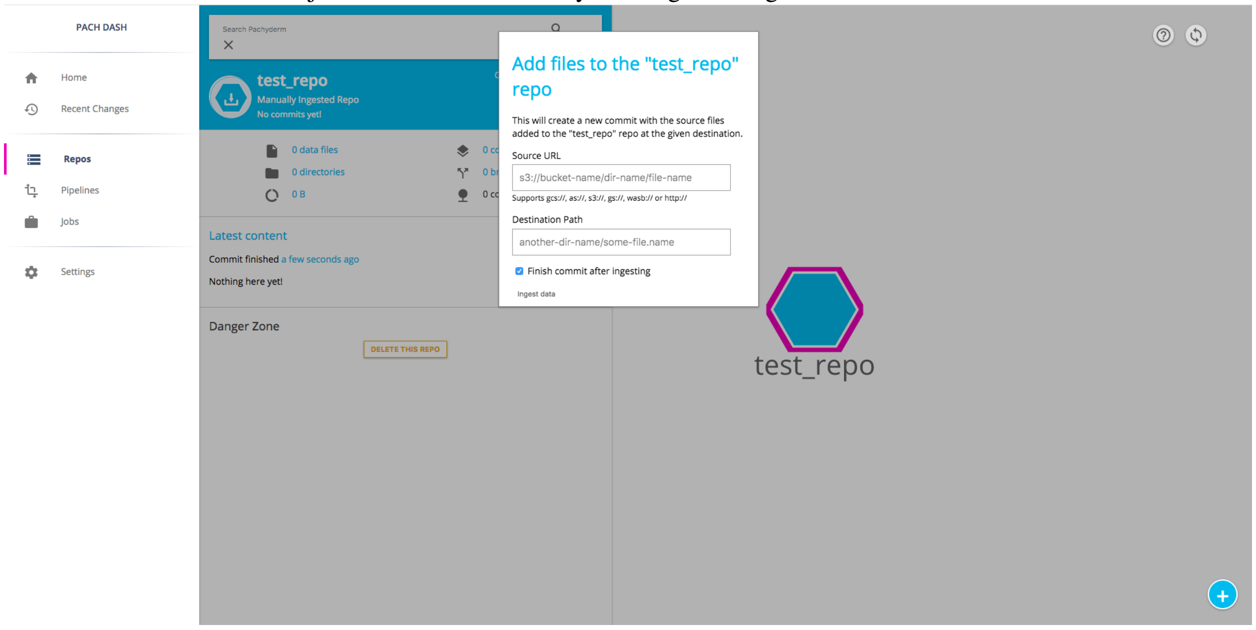
Once you fill in your name and click save, the new data repository will show up in the main dashboard screen:



To add data to this repository, you can click on the blue icon representing the repo. This will present you with some details about the repo along with an "ingest data" icon:



You can add data from an object store or other URL by clicking this "ingest data" icon:



Add files to the "test_repo" repo

This will create a new commit with the source files added to the "test_repo" repo at the given destination.

Source URL

s3://bucket-name/dir-name/file-name

Supports gcs://, as://, s3://, gs://, wasb:// or http://

Destination Path

another-dir-name/some-file.name

☑ Finish commit after ingesting

Ingest data

CHAPTER 5

Creating Analysis Pipelines

There are three steps to running an analysis in a Pachyderm "pipeline":

1. Write your code.
2. Build a Docker image that includes your code and dependencies.
3. Create a Pachyderm "pipeline" referencing that Docker image.

Multi-stage pipelines (e.g., parsing -> modeling -> output) can be created by repeating these three steps to build up a graph of processing steps.

5.1 1. Writing your analysis code

Code used to process data in Pachyderm can be written using any languages or libraries you want. It can be as simple as a bash command or as complicated as a TensorFlow neural network. At the end of the day, all your code and dependencies will be built into a container that can run anywhere (including inside of Pachyderm). We've got demonstrative examples on GitHub using bash, Python, TensorFlow, and OpenCV and we're constantly adding more.

As we touch on briefly in the beginner tutorial, your code itself only needs to read and write files from a local file system. It does NOT have to import any special Pachyderm functionality or libraries. You just need to be able to read files and write files.

For the reading files part, Pachyderm automatically mounts each input data repository as `/pfs/<repo_name>` in the running instances of your Docker image (called "containers"). The code that you write just needs to read input data from this directory, just like in any other file system. Your analysis code also does NOT have to deal with data sharding or parallelization as Pachyderm will automatically shard the input data across parallel containers. For example, if you've got four containers running your Python code, Pachyderm will automatically supply 1/4 of the input data to `/pfs/<repo_name>` in each running container. That being said, you also have a lot of control over how that input data is split across containers. Check out our guide on parallelism and distributed computing for more details on that subject.

For the writing files part (saving results, etc.), your code simply needs to write to `/pfs/out`. This is a special directory mounted by Pachyderm in all of your running containers. Similar to reading data, your code doesn't have to manage parallelization or sharding, just write data to `/pfs/out` and Pachyderm will make sure it all ends up in the correct place.

5.2 2. Building a Docker Image

When you create a Pachyderm pipeline (which will be discussed next), you need to specify a Docker image including the code or binary you want to run. Please refer to the official documentation to learn how to build a Docker images.

Note: You specify what commands should run in the container in your pipeline specification (see **Creating a Pipeline** below) rather than the `CMD` field of your Dockerfile, and Pachyderm runs that command inside the container during jobs rather than relying on Docker to run it. The reason is that Pachyderm can't execute your code immediately when your container starts, so it runs a shim process in your container instead, and then calls your pipeline specification's `cmd` from there.

Unless Pachyderm is running on the same host that you used to build your image, you'll need to use a public or private registry to get your image into the Pachyderm cluster. One (free) option is to use Docker's DockerHub registry. You can refer to the official documentation to learn how to push your images to DockerHub. That being said, you are more than welcome to use any other public or private Docker registry.

Note, it is best practice to uniquely tag your Docker images with something other than `:latest`. This allows you to track which Docker images were used to process which data, and will help you as you update your pipelines. You can also utilize the `--build` or `--push-images` flags on `update pipeline` to help you tag your images as they are updated. See the updating pipelines docs for more information.

5.3 3. Creating a Pipeline

Now that you've got your code and image built, the final step is to tell Pachyderm to run the code in your image on certain input data. To do this, you need to supply Pachyderm with a JSON pipeline specification. There are four main components to a pipeline specification: name, transform, parallelism and input. Detailed explanations of the specification parameters and how they work can be found in the pipeline specification docs.

Here's an example pipeline spec:

```
{
  "pipeline": {
    "name": "wordcount"
  },
  "transform": {
    "image": "wordcount-image",
    "cmd": ["/binary", "/pfs/data", "/pfs/out"]
  },
  "input": {
      "pfs": {
        "repo": "data",
        "glob": "/*"
      }
  }
}
```

After you create the JSON pipeline spec (and save it, e.g., as `your_pipeline.json`), you can create the pipeline in Pachyderm using `pachctl`:

```
$ pachctl create pipeline -f your_pipeline.json
```

(`-f` can also take a URL if your JSON manifest is hosted on GitHub or elsewhere. Keeping pipeline specifications under version control is a great idea so you can track changes and seamlessly view or deploy older pipelines if needed.)

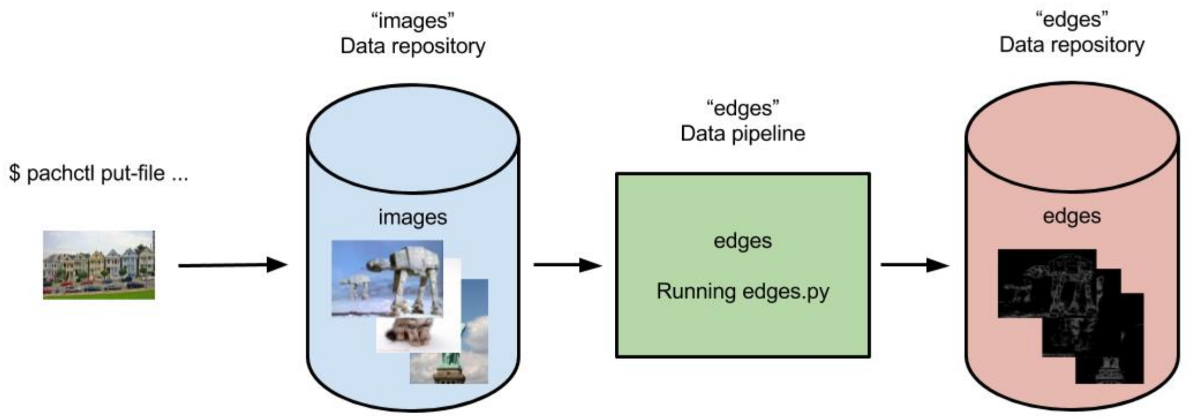
Creating a pipeline tells Pachyderm to run the `cmd` (i.e., your code) in your `image` on the data in the HEAD (most recent) commit of the input repo(s) as well as *all future commits* to the input repo(s). You can think of this pipeline as being "subscribed" to any new commits that are made on any of its input repos. It will automatically process the new data as it comes in.

As soon as you create your pipeline, Pachyderm will launch worker pods on Kubernetes. These worker pods will remain up and running, such that they are ready to process any data committed to their input repos. This allows the pipeline to immediately respond to new data when it's committed without having to wait for their pods to "spin up". However, this has the downside that pods will consume resources even while there's no data to process. You can trade-off the other way by setting the `standby` field to true in your pipeline spec. With this field set, the pipelines will "spin down" when there is no data to process, which means they will consume no resources. However, when new data does come in, the pipeline pods will need to spin back up, which introduces some extra latency. Generally speaking, you should default to not setting standby until cluster utilization becomes a concern. When it does, pipelines that run infrequently and are highly parallel are the best candidates for `standby` .

Getting Data Out of Pachyderm

Once you've got one or more pipelines built and have data flowing through Pachyderm, you need to be able to track that data flowing through your pipeline(s) and get results out of Pachyderm. Let's use the OpenCV pipeline as an example.

Here's what our pipeline and the corresponding data repositories look like:



Every commit of new images into the "images" data repository results in a corresponding output commit of results into the "edges" data repository. But how do we get our results out of Pachyderm? Moreover, how would we get the particular result corresponding to a particular input image? That's what we will explore here.

6.1 Getting files with `pachctl`

The `pachctl` CLI tool command `get file` can be used to get versioned data out of any data repository:

```
pachctl get file <repo>@<branch-or-commit>:<path/to/file>
```

In the case of the OpenCV pipeline, we could get out an image named `example_pic.jpg`:

```
pachctl get file edges@master:example_pic.jpg
```

But how do we know which files to get? Of course we can use the `pachctl list file` command to see what files are available. But how do we know which results are the latest, came from certain input, etc.? In this case, we would like to know which edge detected images in the `edges` repo come from which input images in the `images` repo. This is where provenance and the `flush commit` command come in handy.

6.2 Examining file provenance with flush commit

Generally, `flush commit` will let our process block on an input commit until all of the output results are ready to read. In other words, `flush commit` lets you view a consistent global snapshot of all your data at a given commit. Note, we are just going to cover a few aspects of `flush commit` here.

Let's demonstrate a typical workflow using `flush commit`. First, we'll make a few commits of data into the `images` repo on the `master` branch. That will then trigger our `edges` pipeline and generate three output commits in our `edges` repo:

```
$ pachctl list commit images
REPO                ID                                  PARENT                              ␣
↪     STARTED              DURATION            SIZE
images              c721c4bb9a8046f3a7319ed97d256bb9  ␣
↪a9678d2a439648c59636688945f3c6b5   About a minute ago   1 seconds            932.2␣
↪KiB
images              a9678d2a439648c59636688945f3c6b5  ␣
↪87f5266ef44f4510a7c5e046d77984a6   About a minute ago   Less than a second   238.3␣
↪KiB
images              87f5266ef44f4510a7c5e046d77984a6   <none>                             ␣
↪     10 minutes ago       Less than a second   57.27 KiB
$ pachctl list commit edges
REPO                ID                                  PARENT                              ␣
↪     STARTED              DURATION            SIZE
edges               f716eabf95854be285c3ef23570bd836  ␣
↪026536b547a44a8daa2db9d25bf88b79   About a minute ago   Less than a second   233.7␣
↪KiB
edges               026536b547a44a8daa2db9d25bf88b79  ␣
↪754542b89c1c47a5b657e60381c06c71   About a minute ago   Less than a second   133.6␣
↪KiB
edges               754542b89c1c47a5b657e60381c06c71   <none>                             ␣
↪     2 minutes ago        Less than a second   22.22 KiB
```

In this case, we have one output commit per input commit on `images`. However, this might get more complicated for pipelines with multiple branches, multiple PFS inputs, etc. To confirm which commits correspond to which outputs, we can use `flush commit`. In particular, we can call `flush commit` on any one of our commits into `images` to see which output came from this particular commit:

```
$ pachctl flush commit images@a9678d2a439648c59636688945f3c6b5
REPO                ID                                  PARENT                              ␣
↪     STARTED             DURATION            SIZE
edges               026536b547a44a8daa2db9d25bf88b79  ␣
↪754542b89c1c47a5b657e60381c06c71   3 minutes ago       Less than a second   133.6␣
↪KiB
```

6.3 Exporting data via `egress`

In addition to getting data out of Pachyderm with `pachctl get file`, you can add an optional `egress` field to your pipeline specification. `egress` allows you to push the results of a Pipeline to an external data store such as S3, Google Cloud Storage or Azure Blob Storage. Data will be pushed after the user code has finished running but before the job is marked as successful.

6.4 Other ways to view, interact with, or export data in Pachyderm

Although `pachctl` and `egress` provide easy ways to interact with data in Pachyderm repos, they are by no means the only ways. For example, you can:

- Have one or more of your pipeline stages connect and export data to databases running outside of Pachyderm.
- Use a Pachyderm service to launch a long running service, like Jupyter, that has access to internal Pachyderm data and can be accessed externally via a specified port.
- Mount versioned data from the distributed file system via `pachctl mount ...` (a feature best suited for experimentation and testing).
- If you're on Pachyderm Enterprise, you can use the s3gateway, which allows you to reuse existing tools or libraries that work with object stores. See the s3gateway docs for more information.

CHAPTER 7

Deleting Data in Pachyderm

Sometimes "bad" data gets committed to Pachyderm and you need a way to delete it. There are a couple of ways to address this, which depend on what exactly was "bad" about the data you committed and what's happened in the system since you committed the "bad" data.

- *Deleting the HEAD of a branch* - You should follow this guide if you've just made a commit to a branch with some corrupt, incorrect, or otherwise bad changes to your data.
- *Deleting non-HEAD commits* - You should follow this guide if you've committed data to the branch after committing the data that needs to be deleted.
- *Deleting sensitive data* - You should follow these steps when you have committed sensitive data that you need to completely purge from Pachyderm, such that no trace remains.

7.1 Deleting The HEAD of a Branch

The simplest case is when you've just made a commit to a branch with some incorrect, corrupt, or otherwise bad data. In this scenario, the HEAD of your branch (i.e., the latest commit) is bad. Users who read from it are likely to be misled, and/or pipeline subscribed to it are likely to fail or produce bad downstream output.

To fix this you should use `delete commit` as follows:

```
$ pachctl delete commit <repo>@<branch-or-commit-id>
```

When you delete the bad commit, several things will happen (all atomically):

- The commit metadata will be deleted.
- Any branch that the commit was the HEAD of will have its HEAD set to the commit's parent. If the commit's parent is `nil`, the branch's HEAD will be set to `nil`.
- If the commit has children (commits which it is the parent of), those children's parent will be set to the deleted commit's parent. Again, if the deleted commit's parent is `nil` then the children commit's parent will be set to `nil`.
- Any jobs which were created due to this commit will be deleted (running jobs get killed). This includes jobs which don't directly take the commit as input, but are farther downstream in your DAG.
- Output commits from deleted jobs will also be deleted, and all the above effects will apply to those commits as well.

7.2 Deleting Non-HEAD Commits

Recovering from commits of bad data is a little more complicated if you've committed more data to the branch after the bad data was added. You can still delete the commit as in the previous section, however, unless the subsequent commits overwrote or deleted the bad data, it will still be present in the children commits. *Deleting a commit does not modify its children.*

In git terms, `delete commit` is equivalent to squashing a commit out of existence. It's not equivalent to reverting a commit. The reason for this behavior is that the semantics of revert can get ambiguous when the files being reverted have been otherwise modified. Git's revert can leave you with a merge conflict to solve, and merge conflicts don't make sense with Pachyderm due to the shared nature of the system and the size of the data being stored.

In these scenario, you can also delete the children commits, however those commits may also have good data that you don't want to delete. If so, you should:

1. Start a new commit on the branch with `pachctl start commit`.
2. Delete all bad files from the newly opened commit with `pachctl delete file`.
3. Finish the commit with `pachctl finish commit`.
4. Delete the initial bad commits and all children up to the newly finished commit.

Depending on how you're using Pachyderm, the final step may be optional. After you finish the "fixed" commit, the HEADs of all your branches will converge to correct results as downstream jobs finish. However, deleting those commits allow you to clean up your commit history and makes sure that no one will ever access errant data when reading non-HEAD version of the data.

7.3 Deleting Sensitive Data

If the data you committed is bad because it's sensitive and you want to make sure that nobody ever accesses it, you should complete an extra step in addition to those above.

Pachyderm stores data in a content addressed way and when you delete a file or a commit, Pachyderm only deletes references to the underlying data, it doesn't delete the actual data until it performs garbage collection. To truly purge the data you must delete all references to it using the methods described above, and then you must run a garbage collect with `pachctl garbage collect`.

CHAPTER 8

Appending vs Overwriting Files

8.1 Introduction

Pachyderm is designed to work with pipelined data processing in a containerized environment. The Pachyderm File System (pfs) is a file-based system that is distributed and supports data of all types of files (binary, csv, json, images, etc) from many sources and users. That data is processed in parallel across many different jobs and pipelines using the Pachyderm Pipeline System (pps). The Pachyderm File System (pfs) and Pachyderm Pipeline System (pps) are designed to work together to get the right version of the right data to the right container at the right time.

Among the many complexities you must consider are these:

- Files can be put into pfs in "append" or "overwrite" mode.
- Pipeline definitions use glob patterns to filter a view of input repositories.
- The Pachyderm Pipeline System must merge data from what may be multiple containers running the same pipeline code, at the same time.

When you add in the ability to do "cross" and "union" operators on multiple input repositories to those three considerations, it can be a little confusing to understand what's actually happening with your files!

This document will take you through some of the advanced details, best practices, and conventions for the following. If you're unfamiliar with the topic, each link below will take you to the basics.

- loading data into Pachyderm,
- using glob patterns to filter the data to your pipelines, and
- processing data with your pipelines and placing it in output repos.

8.2 Loading data into Pachyderm

8.2.1 Appending to files

When putting files into a pfs repo via Pachyderm's `pachctl` utility or via the Pachyderm APIs, it's vital to know about the default behaviors of the `put file` command. The following commands create the repo "voterData" and place a local file called "OHVoterData.csv" into it.

```
$ pachctl create repo voterData
$ pachctl put file voterData@master -f OHVoterData.csv
```

The file will, by default, be placed into the top-level directory of voterData with the name "OHVoterData.csv". If the file were 153.8KiB, running the command to list files in that repo would result in

```
$ pachctl list file voterData@master
COMMIT                           NAME             TYPE COMMITTED    SIZE
8560235e7d854eae80aa03a33f8927eb /OHVoterData.csv file 1 second ago 153.8KiB
```

If you were to re-run the `put file` command above, by default, the file would be appended to itself and listing the repo would look like this:

```
$ pachctl list file voterData@master
COMMIT                           NAME             TYPE COMMITTED     SIZE
105aab526f064b58a351fe0783686c54 /OHVoterData.csv file 2 seconds ago 307.6KiB
```

In this case, any pipelines that use this repo for input will see an updated file that has double the data in it. It may also have an intermediate header row. (See Specifying Header/Footer for details on headers and footers in files.) This is Pachyderm's default behavior. What if you want to overwrite the files?

8.2.2 Overwriting files

This is where the `-o` (or `--overwrite`) flag comes in handy. It will, as you've probably guessed, overwrite the file, rather than append it.

```
$ pachctl put file voterData@master -f OHVoterData.csv --overwrite
$ pachctl list file voterData@master
COMMIT                           NAME             TYPE COMMITTED    SIZE
8560235e7d854eae80aa03a33f8927eb /OHVoterData.csv file 1 second ago 153.8KiB
$ pachctl put file voterData@master -f OHVoterData.csv --overwrite
$ pachctl list file voterData@master
COMMIT                           NAME             TYPE COMMITTED    SIZE
4876f99951cc4ea9929a6a213554ced8 /OHVoterData.csv file 1 second ago 153.8KiB
```

8.2.3 Deduplication

Pachyderm will deduplicate data loaded into input repositories. If you were to load another file that hashed identically to "OHVoterData.csv", there would be one copy of the data in Pachyderm with two files' metadata pointing to it. This works even when using the append behavior above. If you were to put a file named OHVoterData2004.csv that was identical to that first `put file` of OHVoterData.csv, and then update OHVoterData.csv as shown above, there would be two sets of bits in Pachyderm:

- a set of bits that would be returned when asking for the old branch of OHVoterData.csv & OHVoterData2004.csv and
- a set of bits that would be appended to that first set to assemble the new, master branch of OHVoterData.csv.

This deduping happens with each file as it's added to Pachyderm. We don't do deduping within the file ("intrafile deduplication") because this system is built to work with any file type.

Important: Pachyderm is smart about keeping the minimum set of bits in the object store and assembling the version of the file you (or your code!) have asked for.

8.2.4 Use cases for large datasets in single files

Splitting large files in Pachyderm

Unlike a system like Git, which expects almost all of your files to be text, Pachyderm does not do intra-file diffing because we work with any file type:

- text
- JSON
- images
- video
- binary
- and so on…

Pachyderm diffs content at the per-file level. Therefore, if one bit in content of a file changes, Pachyderm sees that as a new file. Similarly, Pachyderm can only distribute computation at the level of a single file; if your data is only one large file, it can only be processed by a single worker.

Because of these reasons, it's pretty common to break up large files into smaller chunks. For simple data types, Pachyderm provides the `--split` flag to `put file` to automatically do this for you. For more complex splitting patterns (e.g. `avro` or other binary formats), you'll need to manually split your data either at ingest or with a Pachyderm pipeline.

Split and target-file flags

For common file types that are often used in data science, such as CSV, line-delimited text files, JavaScript Object Notation (json) files, Pachyderm includes the powerful `--split` , `--target-file-bytes` and `--target-file-datums` flags.

`--split` will divide those files into chunks based on what a "record" is. In line-delimited files, it's a line. In json files, it's an object. `--split` takes one argument: `line` , `json` or `sql` .

Note: See the Splitting Data for Distributed Processing cookbook for more details on SQL support.

This argument tells Pachyderm how you want the file split into chunks. For example, if you use `--split line` , Pachyderm will only divide your file on newline boundaries, never in the middle of a line. Along with the `--split` flag, it's common to use additional "target" flags to get better control over the details of the split.

Note: We'll call each of the chunks a "split-file" in this document.

- `--target-file-bytes` will fill each of the split-files with data up to the number of bytes you specify, splitting on the nearest record boundary. Let's say you have a line-delimited file of 50 lines, with each line having about 20 bytes. If you use the flags `--split lines --target-file-bytes 100` , you'll see the input file split into about 10 files or so, each of which will have 5 or so lines. Each split-file's size will hover above the target value of 100 bytes, not going below 100 bytes until the last split-file, which may be less than 100 bytes.
- `--target-file-datums` will attempt to fill each split-file with the number of datums you specify. Going back to that same line-delimited 50-line file above, if you use `--split lines --target-file-datums 2` , you'll see the file split into 50 split-files, each of which will have 2 lines.

- Specifying both flags, `--target-file-datums` and `--target-file-bytes`, will result in each split-file containing just enough data to satisfy whichever constraint is hit first. Pachyderm will split the file and then fill the first target split-file with line-based records until it hits the record limit. If it passes the target byte number with just one record, it will move on to the next split-file. If it hits the target datum number after adding another line, it will move on to the next split-file. Using the example above, if the flags supplied to `put file` are `--split lines --target-file-datums 2 --target-file-bytes 100`, it will have the same result as `--target-file-datums 2`, since that's the most compact constraint, and file sizes will hover around 40 bytes.

What split data looks like in a Pachyderm repository

Going back to our 50-line file example, let's say that file is named "my-data.txt". We'll create a repo named "line-data" and load my-data.txt into Pachyderm with the following commands:

```
$ pachctl create repo line-data
$ pachctl put file line-data@master -f my-data.txt --split line
```

After `put file` is complete, list the files in the repo.

```
$ pachctl list file line-data@master
COMMIT                           NAME         TYPE COMMITTED          SIZE
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt dir  About a minute ago 1.071KiB
```

Important: The `list file` command indicates that the line-oriented file we uploaded, "my-data.txt", is actually a directory.

This file *looks* like a directory because the `--split` flag has instructed Pachyderm to split the file up, and it has created a directory with all the chunks in it. And, as you can see below, each chunk will be put into a file. Those are the split-files. Each split-file will be given a 16-character filename, left-padded with 0.

Note: `--split` does not currently allow you to define more sophisticated file names. This is a set of features we'll add in future releases. (See Issue 3568, for example).

Each filename will be numbered sequentially in hexadecimal. We modify the command to list the contents of "my-data.txt", and the output reveals the naming structure used:

```
$ pachctl list file line-data@master my-data.txt
COMMIT                           NAME                           TYPE COMMITTED         ␣
↪ SIZE
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000000 file About a minute␣
↪ago 21B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000001 file About a minute␣
↪ago 22B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000002 file About a minute␣
↪ago 24B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000003 file About a minute␣
↪ago 21B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000004 file About a minute␣
↪ago 22B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000005 file About a minute␣
↪ago 24B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000006 file About a minute␣
↪ago 21B
```

```
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000007 file About a minute␣
↪ago 22B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000008 file About a minute␣
↪ago 23B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000009 file About a minute␣
↪ago 24B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/000000000000000a file About a minute␣
↪ago 24B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/000000000000000b file About a minute␣
↪ago 24B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/000000000000000c file About a minute␣
↪ago 21B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/000000000000000d file About a minute␣
↪ago 23B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/000000000000000e file About a minute␣
↪ago 21B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/000000000000000f file About a minute␣
↪ago 21B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000010 file About a minute␣
↪ago 21B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000011 file About a minute␣
↪ago 22B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000012 file About a minute␣
↪ago 21B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000013 file About a minute␣
↪ago 23B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000014 file About a minute␣
↪ago 21B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000015 file About a minute␣
↪ago 21B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000016 file About a minute␣
↪ago 24B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000017 file About a minute␣
↪ago 22B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000018 file About a minute␣
↪ago 23B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000019 file About a minute␣
↪ago 21B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/000000000000001a file About a minute␣
↪ago 21B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/000000000000001b file About a minute␣
↪ago 22B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/000000000000001c file About a minute␣
↪ago 22B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/000000000000001d file About a minute␣
↪ago 21B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/000000000000001e file About a minute␣
↪ago 22B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/000000000000001f file About a minute␣
↪ago 21B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000020 file About a minute␣
↪ago 21B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000021 file About a minute␣
↪ago 21B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000022 file About a minute␣
↪ago 21B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000023 file About a minute␣
↪ago 22B
```

```
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000024 file About a minute␣
↪ago 21B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000025 file About a minute␣
↪ago 23B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000026 file About a minute␣
↪ago 21B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000027 file About a minute␣
↪ago 21B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000028 file About a minute␣
↪ago 24B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000029 file About a minute␣
↪ago 22B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/000000000000002a file About a minute␣
↪ago 23B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/000000000000002b file About a minute␣
↪ago 21B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/000000000000002c file About a minute␣
↪ago 21B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/000000000000002d file About a minute␣
↪ago 22B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/000000000000002e file About a minute␣
↪ago 22B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/000000000000002f file About a minute␣
↪ago 21B
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000030 file About a minute␣
↪ago 22B
COMMIT                           NAME                           TYPE COMMITTED          ␣
↪ SIZE
8cce4de3571f46459cbe4d7fe222a466 /my-data.txt/0000000000000031 file About a minute␣
↪ago 22B
```

Appending to files with –split

Combining `--split` with the default "append" behavior of `pachctl put file` allows flexible and scalable processing of record-oriented file data from external, legacy systems. Each of the split-files will be deduplicated. You would have to ensure that `put file` commands always have the `--split` flag.

`pachctl` will reject the command if `--split` is not specified to append a file that it was previously specified with an error like this

```
could not put file at "/my-data.txt"; a file of type directory is already there
```

Pachyderm will ensure that only the added data will get reprocessed when you append to a file using `--split`. Each of the split-files is subject to deduplication, so storage will be optimized. A large file with many duplicate lines (or objects that hash identically) which you with `--split` may actually take up less space in pfs than it does as a single file outside of pfs.

Appending files can make for efficient processing in downstream pipelines. For example, let's say you have a file named "count.txt" consisting of 5 lines

```
One
Two
Three
Four
Five
```

Loading that local file into Pachyderm using `--split` with a command like

```
pachctl put file line-data@master:count.txt -f ./count.txt --split line
```

will result in five files in a directory named "count.txt" in the input repo, each of which will have the following contents

```
count.txt/0000000000000000: One
count.txt/0000000000000001: Two
count.txt/0000000000000002: Three
count.txt/0000000000000003: Four
count.txt/0000000000000004: Five
```

This would result in five datums being processed in any pipelines that use this repo.

Now, take a one-line file containing

```
Six
```

and load it into Pachyderm appending it to the count.txt file. If that file were named, "more-count.txt", the command might look like

```
pachctl put file line-data@master:my-data.txt -f more-count.txt --split line
```

That will result in six files in the directory named "count.txt" in the input repo, each of which will have the following contents

```
count.txt/0000000000000000: One
count.txt/0000000000000001: Two
count.txt/0000000000000002: Three
count.txt/0000000000000003: Four
count.txt/0000000000000004: Five
count.txt/0000000000000005: Six
```

This would result in one datum being processed in any pipelines that use this repo: the new file `count.txt/0000000000000005`.

Overwriting files with –split

The behavior of Pachyderm when a file loaded with `--split` is overwritten is simple to explain but subtle in its implications. Remember that the loaded file will be split into those sequentially-named files, as shown above. If any of those resulting split-files hashes differently than the one it's replacing, that will cause the Pachyderm Pipeline System to process that data.

This can have important consequences for downstream processing. For example, let's say you have that same file named "count.txt" consisting of 5 lines that we used in the previous example

```
One
Two
Three
Four
Five
```

As discussed prior, loading that file into Pachyderm using `--split` will result in five files in a directory named "count.txt" in the input repo, each of which will have the following contents

```
count.txt/0000000000000000: One
count.txt/0000000000000001: Two
count.txt/0000000000000002: Three
count.txt/0000000000000003: Four
count.txt/0000000000000004: Five
```

This would result in five datums being processed in any pipelines that use this repo.

Now, modify that file by inserting the word "Zero" on the first line.

```
Zero
One
Two
Three
Four
Five
```

Let's upload it to Pachyderm using `--split` and `--overwrite`.

```
pachctl put file line-data@master:count.txt -f ./count.txt --split line --overwrite
```

The input repo will now look like this

```
count.txt/0000000000000000: Zero
count.txt/0000000000000001: One
count.txt/0000000000000002: Two
count.txt/0000000000000003: Three
count.txt/0000000000000004: Four
count.txt/0000000000000005: Five
```

As far as Pachyderm is concerned, every single file existing has changed, and a new file has been added. This is because the filename is taken into account when hashing the data for the pipeline. While only one new piece of content is being stored, `Zero`, all six datums would be processed by a downstream pipeline.

It's important to remember that what looks like a simple upsert can be a kind of a fencepost error. Being "off by one line" in your data can be expensive, consuming processing resources you didn't intend to spend.

8.2.5 Datums in Pachyderm pipelines

The "datum" is the fundamental unit of data processing in Pachyderm pipelines. It is defined at the file level and filtered by the "globs" you specify in your pipelines. *What* makes a datum is defined by you. How do you do that?

When creating a pipeline, you can specify one or more input repos. Each of these will contain files. Those files are filtered by the "glob" you specify in the pipeline's definition, along with the input operators you use. That determines how the datums you want your pipeline to process appear in the pipeline: globs and input operators, along with other pipeline configuration operators, specify how you would like those datums orchestrated across your processing containers. Pachyderm Pipeline System (pps) processes each datum individually in containers in pods, using Pachyderm File System (pfs) to get the right data to the right code at the right time and merge the results.

To summarize:

- **repos** contain *files* in pfs
- **pipelines** filter and organize those files into *datums* for processing through *globs* and *input repo operators*
- pps will use available resources to process each datum, using pfs to assign datums to containers and merge results in the pipeline's output repo.

Let's start with one of the simplest pipelines. The pipeline has a single input repo, `my-data`. All it does is copy data from its input to its output.

```
{
  "pipeline": {
    "name": "my-pipeline"
  },
  "input": {
    "pfs": {
      "glob": "/*",
      "repo": "my-data"
    }
  },
  "transform": {
      "cmd": ["sh"],
      "stdin": ["/bin/cp -r /pfs/my-data/\* /pfs/out/"],
    "image": "ubuntu:14.04"
  }
}
```

With this configuration, the `my-pipeline` repo will always be a copy of the `my-data` repo. Where it gets interesting is in the view of jobs processed. Let's say you have two data files and you use the `put file` command to load both of those into my-data

```
$ pachctl put file my-data@master -f my-data-file-1.txt -f my-data-file-2.txt
```

Listing jobs will show that the job had 2 input datums, something like this:

```
$ pachctl list job
ID                               PIPELINE    STARTED        DURATION           ␣
↪RESTART PROGRESS  DL       UL       STATE
0517ff33742a4fada32d8d43d7adb108 my-pipeline 20 seconds ago Less than a second 0       ␣
↪ 2 + 0 / 2 3.218KiB 3.218KiB success
```

What if you had defined the pipeline to use the "/" glob, instead? That `list job` output would've showed one datum, because it treats the entire input directory as one datum.

```
$ pachctl list job
ID                               PIPELINE    STARTED        DURATION           ␣
↪RESTART PROGRESS  DL       UL       STATE
aa436dbb53ba4cee9baaf84a1cc6717a my-pipeline 19 seconds ago Less than a second 0       ␣
↪ 1 + 0 / 1 3.218KiB 3.218KiB success
```

If we had written that pipeline to have a `parallelism_spec` of greater than 1, there would have still been only one pod used to process that data. You can find more detailed information on how to use Pachyderm Pipeline System and globs to do sophisticated configurations in the Distributed Computing section of our documentation.

When you have loaded data via a `--split` flag, as discussed above, you can use the glob to select the split-files to be sent to a pipeline. A detailed discussion of this is available in the Pachyderm cookbook section Splitting Data for Distributed Processing.

8.2.6 Summary

Pachyderm provides powerful operators for combining and merging your data through input operations and the glob operator. Each of these have subtleties that are worth working through with concrete examples.

CHAPTER 9

Lifecycle of a Datum

9.1 Introduction

Pachyderm's idea of a "datum" in the context of parallel processing has subtle implications for how data is written to output files. It's important to understand under what conditions data will be overwritten or merged in output files.

There are four basic rules to how Pachyderm will process your data in the pipelines you create.

- Pachyderm will split your input into individual datums as you specified in your pipeline spec
- each datum will be processed independently, using the parallelism you specified in your pipeline spec, with no guarantee of the order of processing
- Pachyderm will merge the output of each pod into each output file, with no guarantee of ordering
- the output files will look as if everything was done in one processing step

If one of your pipelines is written to take all its input, whatever it may be, process it, and put the output into one file, the final output will be one file.
The many datums in your pipeline's input would become one file at the end, with the processing of every datum reflected in that file.

If you write a pipeline to write to multiple files, each of those files may contained merged data that looks as if it were all processed in one step, even if you would have specified the pipeline to process each datum in one container via a `parallelism_spec` set to the default value of 1.

This sounds a little complicated, but it breaks down simply into three distinct relationships between the datums in your pipeline's input repository and in its output repository. We'll list them, and then go into them in detail, pointing to examples that illustrate them.

- 1 to 1: 1 unique datum in to 1 unique datum out
- 1 to many: 1 unique datum in to many unique datums out
- Many to many: many unique datums in map to many datums out.

9.1.1 1:1

The best example of this is the opencv example and beginner's tutorial.

One datum, in this case an image, is transformed into another datum, in this case, another image.

9.1.2 1:many or 1:N

A good example of this is a pipeline designed to split an image into many tiles for further analysis, an easy extension of the opencv example, left as an exercise for you. Each tile is unique in the output by necessity; you can't have one image stomping on another image's tile!

9.1.3 Many:Many or N:M

This is the most general case of a Pachyderm pipeline: it takes many unique datums in and may write to many output files for each one. Those output files may not be unique across datums. That means that the results from processing many datums may be represented in a single file. One of the things Pachyderm does is merge all those results into the single file for you. This is what "merging" in the output from `pachctl list jobs` means.

To visualize what the many-to-many case looks like, we'll use the wordcount example. (We'll also use a version of wordcount later on in this document.) The illustration below shows what this might look like in action by showing a subset of the wordcount pipeline's output datums. In this example, after processing the first paragraph of each book, *Moby Dick* (`md.txt`) and *A Tale of Two Cities* (`ttc.txt`), we show ten example output wordcount files.

The words `was`, `best`, `worst`, `wisdom`, and `foolishness` are unique to *A Tale of Two Cities* and thus each eponymous file solely contains output from processing `ttc.txt`.

The words `Ishmael` and `money` are unique to *Moby Dick*, thus the files each solely contain output from processing `md.txt`.

The files `it`, `the`, and `of` are not unique to each text. The pipeline will output to the word file in each container and Pachyderm will handle merging the results, as shown in the image.

Important: The order of results isn't guaranteed.

The Pachyderm Pipeline System works with the Pachyderm File System to make sure that files output by each pipeline are merged successfully at the end of every commit.

9.2 Cross and union inputs

When creating pipelines, you can use "union" and "cross" operations to combine inputs.

Union input will combine each of the datums in the input repos as one set of datums. The result is that the number of datums processed is the sum of all the datums in each repo.

For example, let's say you have two input repos, A and B. Each of them contain three files with the same names: 1.txt, 2.txt, and 3.txt. Each file hashes differently, because each of the files contains different content, despite the identical filenames. If you were to cross them in a pipeline, the "input" object in the pipeline spec might look like this.

```
"input": {
    "union": [
        {
            "pfs": {
                "glob": "/*",
                "repo": "A"
            }
        },
        {
            "pfs": {
                "glob": "/*",
                "repo": "B"
            }
        }
    ]
}
```

If each repo had those three files at the top, there would be six (6) datums overall, which is the sum of the number of input files. You'd see the following datums, in a random order, in your pipeline as it ran though them.

```
/pfs/A/1.txt

/pfs/A/2.txt

/pfs/A/3.txt

/pfs/B/1.txt

/pfs/B/2.txt

/pfs/B/3.txt
```

This is shown in the animation below. Remember that the order in which the datums will be processed is not guaranteed; we'll show them in a random order.

One of the ways you can make your code simpler is to use the `name` field for the `pfs` object and give each of those repos the same name.

```
"input": {
    "union": [
        {
            "pfs": {
                "name": "C",
                "glob": "/*",
                "repo": "A"
            }
        },
        {
            "pfs": {
                "name": "C",
                "glob": "/*",
                "repo": "B"
            }
        }
    ]
}
```

You'd still see the same six datums, in a random order, in your pipeline as it ran though them, but they'd all be in a directory with the same name: C.

```
/pfs/C/1.txt  # from A

/pfs/C/2.txt  # from A

/pfs/C/3.txt  # from A

/pfs/C/1.txt  # from B

/pfs/C/2.txt  # from B

/pfs/C/3.txt  # from B
```

This is shown in the animation below. Remember that the order in which the datums will be processed is not guaranteed, so we show them in a random order. We also highlight the input file at the top of the image, so it's easier to understand when a file from a particular repo is being processed.

Cross input is a cross-product of all the datums, selected by the globs on the repos you're crossing. It provides a combination of all the datums to the pipeline that uses it as input, treating each combination as a datum.

There are many examples that show the power of this operator: Combining/Merging/Joining Data cookbook and the Distributed hyperparameter tuning example are good ones.

It's important to remember that paths in your input repo will be preserved and prefixed by the repo name to prevent collisions between identically-named files. For example, let's take those same two input repos, A and B, each of which with the same files as above. If you were to cross them in a pipeline, the "input" object in the pipeline spec might look like this

```
"input": {
    "cross": [
        {
            "pfs": {
                "glob": "/*",
                "repo": "A"
            }
        },
        {
            "pfs": {
                "glob": "/*",
                "repo": "B"
            }
        }
    ]
}
```

In the case of cross inputs, you can't give the repos being crossed identical names, because of that need to avoid name collisions. If each repo had those three files at the top, there would be nine (9) datums overall, which is every permutation of the input files. You'd see the following datums, in a random order, in your pipeline as it ran though the nine permutations.

```
/pfs/A/1.txt
/pfs/B/1.txt

/pfs/A/2.txt
/pfs/B/1.txt

/pfs/A/3.txt
```

```
/pfs/B/1.txt

/pfs/A/1.txt
/pfs/B/2.txt

/pfs/A/2.txt
/pfs/B/2.txt

/pfs/A/3.txt
/pfs/B/2.txt

/pfs/A/1.txt
/pfs/B/3.txt

/pfs/A/1.txt
/pfs/B/2.txt

/pfs/A/1.txt
/pfs/B/3.txt
```

Important: You (or your code) will always see *both* input directories involved in the cross!

9.3 Output repositories

Every Pachyderm pipeline has an output repository associated with it, with the same name as the pipeline. For example, an "edges" pipeline would have an "edges" repository you can use as input to other pipelines, like a "montage" pipeline.

Your code, regardless of the pipeline you put it in, should place data in a filesystem mounted under "/pfs/out" and it will appear in the named repository for the current pipeline.

9.3.1 Appending vs overwriting data in output repositories

The Pachyderm File System keeps track of which datums are being processed in which containers, and makes sure that each datum leaves its unique data in output files. Let's say you have a simple pipeline, "wordcount", that counts references to words in documents by writing the number of occurrences of a word to an output file named for each word in `/pfs/out`, followed by a newline. We intend to process the data by treating each input file as a datum. We specify the glob in the "wordcount" pipeline json accordingly, something like `"glob": "/*"`. We load a file containing the first paragraph of Charles Dickens's "A Tale of Two Cities" into our input repo, but mistakenly just put the first four lines in the file `ttc.txt`.

```
It was the best of times,
it was the worst of times,
it was the age of wisdom,
it was the age of foolishness,
```

In this case, after the pipeline runs on this single datum, `/pfs/out` would contain the files

```
it -> 4\n
was -> 4\n
the -> 4\n
best -> 1\n
```

```
worst -> 1\n
of -> 4\n
times -> 2\n
age -> 2\n
wisdom -> 1\n
foolishness -> 1\n
```

Where `\n` is the newline appended by our "wordcount" code after it outputs the word count. If we were to fix `ttc.txt`, either by appending the missing text or replacing it with the entire first paragraph using `pachctl put file` with the `--overwrite` flag, the file would then look like this

```
It was the best of times,
it was the worst of times,
it was the age of wisdom,
it was the age of foolishness,
it was the epoch of belief,
it was the epoch of incredulity,
it was the season of Light,
it was the season of Darkness,
it was the spring of hope,
it was the winter of despair,
we had everything before us,
we had nothing before us,
we were all going direct to Heaven,
we were all going direct the other way--
in short, the period was so far like the present period, that some of
its noisiest authorities insisted on its being received, for good or for
evil, in the superlative degree of comparison only.
```

We would see each file in the "wordcount" repo overwritten with one line with an updated number. Using our existing examples, we'd see a few of the files replaced with new content

```
it -> 10\n
was -> 10\n
the -> 14\n
best -> 1\n
worst -> 1\n
of -> 4\n
times -> 2\n
age -> 2\n
wisdom -> 1\n
foolishness -> 1\n
```

The reason that the entire file gets reprocessed, even if we just append to it, is because the entire file is the datum. We haven't used the `--split` flag combined with the appropriate glob to split it into lots of datums.

What if we put other texts in the pipeline's input repo? Such as the first paragraph of Herman Melville's Moby Dick, put into "md.txt".

```
Call me Ishmael. Some years ago--never mind how long precisely--having
little or no money in my purse, and nothing particular to interest me
on shore, I thought I would sail about a little and see the watery part of the world.
It is a way I have of driving off the spleen and
regulating the circulation. Whenever I find myself growing grim about
the mouth; whenever it is a damp, drizzly November in my soul; whenever
I find myself involuntarily pausing before coffin warehouses, and
bringing up the rear of every funeral I meet; and especially whenever
my hypos get such an upper hand of me, that it requires a strong moral
```

```
principle to prevent me from deliberately stepping into the street, and
methodically knocking people's hats off--then, I account it high time to
get to sea as soon as I can. This is my substitute for pistol and ball.
With a philosophical flourish Cato throws himself upon his sword; I
quietly take to the ship. There is nothing surprising in this. If they
but knew it , almost all men in their degree, some time or other,
cherish very nearly the same feelings towards the ocean with me.
```

What happens to our word files? Will they *all* get overwritten? Not as long as each input file –`ttc.txt` and `md.txt` – is being treated as a separate datum. Only files that contain words that are common between the text will change, because only the added datum will get processed.

You'll see the data in the "wordcount" repo looking something like this:

```
it -> 10\n5\n
was -> 10\n
the -> 14\n7\n
best -> 1\n
worst -> 1\n
of -> 4\n4\n
times -> 2\n
age -> 2\n
wisdom -> 1\n
foolishness -> 1\n
```

During each job that is run, each distinct datum in Pachyderm will put data in an output file. If the file shares a name with the files from other datums, the previously-computed output from each other datum is merged with the new output after processing the new datum. This will happen during the appropriately-named *merge* stage after your pipeline runs. You should not count on the data appearing in a particular order. Before that merge occurs, when your pipeline code is running, you shouldn't expect an output file in the pipeline's repo have any other data in it other than the data from processing that single datum. You won't see it in the output file until all datums have been processed and the merge is complete, after that pipeline and the commit is finished.

What happens if we delete `md.txt` ? The "wordcount" repo would go back to its condition with just `ttc.txt` .

```
it -> 10\n
was -> 10\n
the -> 14\n
best -> 1\n
worst -> 1\n
of -> 4\n
times -> 2\n
age -> 2\n
wisdom -> 1\n
foolishness -> 1\n
```

What if didn't delete `md.txt` ; we appended to it? Then we'd see the appropriate counts change only on the lines of the files affected by `md.txt` ; the counts for `ttc.txt` would not change. Let's say we append the second paragraph to `md.txt` :

```
There now is your insular city of the Manhattoes, belted round by
wharves as Indian isles by coral reefs--commerce surrounds it with her
surf. Right and left, the streets take you waterward. Its extreme
downtown is the battery, where that noble mole is washed by waves, and
cooled by breezes, which a few hours previous were out of sight of
land. Look at the crowds of water-gazers there.
```

The “wordcount” repo might now look like this. (We’re not using stemmed parser, and “it” is a different word than “its”)

```
it -> 10\n6\n
was -> 10\n
the -> 14\n11\n
best -> 1\n
worst -> 1\n
of -> 4\n8\n
times -> 2\n
age -> 2\n
wisdom -> 1\n
foolishness -> 1\n
```

Pachyderm is smart enough to keep track of what changes to what datums affect what downstream results, and only reprocesses and re-merges as needed.

9.4 Summary

To summarize,

- Your output datums should always reflect the state of processing all the input datums in your HEAD commit, independent of whether those input datums were added in separate commits or all added at once.
- If your downstream pipeline processes multiple input datums, putting the result a single file, adding or removing an input datum will only remove its effect from that file. The effect of the other datums will still be seen in that file.

You can see this in action in the word count example in the Pachyderm git repo.

CHAPTER 10

Updating Pipelines

During development, it's very common to update pipelines, whether it's changing your code or just cranking up parallelism. For example, when developing a machine learning model you will likely need to try out a bunch of different versions of your model while your training data stays relatively constant. This is where `update pipeline` comes in.

10.1 Updating your pipeline specification

In cases in which you are updating parallelism, adding another input repo, or otherwise modifying your pipeline specification, you just need to update your JSON file and call `update pipeline`:

```
$ pachctl update pipeline -f pipeline.json
```

Similar to `create pipeline`, `update pipeline` with the `-f` flag can also take a URL if your JSON manifest is hosted on GitHub or elsewhere.

10.2 Updating the code used in a pipeline

You can also use `update pipeline` to update the code you are using in one or more of your pipelines. To update the code in your pipeline:

1. Make the code changes.
2. Build, tag, and push the image in docker to the place specified in the pipeline spec.
3. Call `pachctl update pipeline` again.

10.3 Building pipeline images within pachyderm

Building, tagging and pushing the image in docker requires a bit of ceremony, so there's a shortcut: the `--build` flag for `pachctl update pipeline`. When used, Pachyderm will do the following:

1. Rebuild the docker image.
2. Tag your image with a new unique name.
3. Push that tagged image to your registry (e.g., DockerHub).
4. Update the pipeline specification that you previously gave to Pachyderm to use the new unique tag.

For example, you could update the Python code used in the OpenCV pipeline via:

```
pachctl update pipeline -f edges.json --build --username <registry user>
```

You'll then be prompted for the password associated with the registry user.

10.3.1 Private registries

`--build` supports private registries as well. Make sure the private registry is specified as part of the pipeline spec, and use the `--registry` flag when calling `pachctl update pipeline --build`.

For example, if you wanted to push the image `pachyderm/opencv` to a registry located at `localhost:5000`, you'd have this in your pipeline spec:

```
"image": "localhost:5000/pachyderm/opencv"
```

And would run this to update the pipeline:

```
pachctl update pipeline -f edges.json --build --registry localhost:5000 --username
↪<registry user>
```

10.4 Re-processing data

As of 1.5.1, updating a pipeline will NOT reprocess previously processed data by default. New data that's committed to the inputs will be processed with the new code and "mixed" with the results of processing data with the previous code. Furthermore, data that Pachyderm tried and failed to process with the previous code due to code erroring will be processed with the new code.

`update pipeline` (without flags) is designed for the situation where your code needs to be fixed because it encountered an unexpected new form of data.

If you'd like to update your pipeline and have that updated pipeline reprocess all the data that is currently in the HEAD commit of your input repos, you should use the `--reprocess` flag. This type of update will automatically trigger a job that reprocesses all of the input data in its current state (i.e., the HEAD commits) with the updated pipeline. Then from that point on, the updated pipeline will continue to be used to process any new input data. Previous results will still be available in via their corresponding commit IDs.

CHAPTER 11

Distributed Computing

Distributing computation across multiple workers is a fundamental part of processing any big data or computationally intensive workload. There are two main questions to think about when trying to distribute computation:

1. *How many workers to spread computation across?*
2. *How to define which workers are responsible for which data?*

11.1 Pachyderm Workers

Before we dive into the above questions, there are a few details you should understand about Pachyderm workers.

Every worker for a given pipeline is an identical pod running the Docker image you specified in the pipeline spec. Your analysis code does not need do anything special to run in a distributed fashion. Instead, Pachyderm will spread out the data that needs to be processed across the various workers and make that data available for your code.

Pachyderm workers are spun up when you create the pipeline and are left running in the cluster waiting for new jobs (data) to be available for processing (committed). This saves having to recreate and schedule the worker for every new job.

11.2 Controlling the Number of Workers (Parallelism)

The number of workers that are used for a given pipeline is controlled by the `parallelism_spec` defined in the pipeline specification.

```
"parallelism_spec": {
  // Exactly one of these two fields should be set
  "constant": int
  "coefficient": double
```

Pachyderm has two parallelism strategies: `constant` and `coefficient`. You should set one of the two corresponding fields in the `parallelism_spec`, and pachyderm chooses a parallelism strategy based on which field is set.

If you set the `constant` field, Pachyderm will start the number of workers that you specify. For example, set `"constant":10` to use 10 workers.

If you set the `coefficient` field, Pachyderm will start a number of workers that is a multiple of your Kubernetes cluster's size. For example, if your Kubernetes cluster has 10 nodes, and you set `"coefficient": 0.5`, Pachyderm will start five workers. If you set it to 2.0, Pachyderm will start 20 workers (two per Kubernetes node).

NOTE: The parallelism_spec is optional and will default to `"coefficient": 1`, which means that it'll spawn one worker per Kubernetes node for this pipeline if left unset.

11.3 Spreading Data Across Workers (Glob Patterns)

Defining how your data is spread out among workers is arguably the most important aspect of distributed computation and is the fundamental idea around concepts like Map/Reduce.

Instead of confining users to just data-distribution patterns such as Map (split everything as much as possible) and Reduce (*all* the data must be grouped together), Pachyderm uses Glob Patterns to offer incredible flexibility in defining your data distribution.

Glob patterns are defined by the user for each `pfs` within the `input` of a pipeline, and they tell Pachyderm how to divide the input data into individual "datums" that can be processed independently.

```
"input": {
    "pfs": {
        "repo": "string",
        "glob": "string",
    }
}
```

That means you could easily define multiple PFS inputs, one with the data highly distributed and another where it's grouped together. You can then join the datums in these PFS inputs via a cross product or union (as shown above) for combined, distributed processing.

```
"input": {
    "cross" or "union": [
        {
            "pfs": {
                "repo": "string",
                "glob": "string",
            }
        },
        {
            "pfs": {
                "repo": "string",
                "glob": "string",
            }
        },
        etc...
    ]
}
```

More information about PFS inputs, unions, and crosses can be found in our Pipeline Specification.

11.3.1 Datums

Pachyderm uses the glob pattern to determine how many "datums" a PFS input consists of. Datums are the unit of parallelism in Pachyderm. That is, Pachyderm attempts to process datums in parallel whenever possible.

If you have two workers and define 2 datums, Pachyderm will send one datum to each worker. In a scenario where there are more datums than workers, Pachyderm will queue up extra datums and send them to workers as they finish processing previous datums.

11.3.2 Defining Datums via Glob Patterns

Intuitively, you should think of the PFS input repo as a file system where the glob pattern is being applied to the root of the file system. The files and directories that match the glob pattern are considered datums.

For example, a glob pattern of just `/` would denote the entire input repo as a single datum. All of the input data would be given to a single worker similar to a typical reduce-style operation.

Another commonly used glob pattern is `/*`. `/*` would define each top level object (file or directory) in the PFS input repo as its own datum. If you have a repo with just 10 files in it and no directory structure, every file would be a datum and could be processed independently. This is similar to a typical map-style operation.

But Pachyderm can do anything in between too. If you have a directory structure with each state as a directory and a file for each city such as:

```
/California
   /San-Francisco.json
   /Los-Angeles.json
   ...
/Colorado
   /Denver.json
   /Boulder.json
   ...
...
```

and you need to process all the data for a given state together, `/*` would also be the desired glob pattern. You'd have one datum per state, meaning all the cities for a given state would be processed together by a single worker, but each state can be processed independently.

If we instead used the glob pattern `/*/*` for the states example above, each `<city>.json` would be its own datum.

Glob patterns also let you take only a particular directory (or subset of directories) as a PFS input instead of the whole input repo. If we create a pipeline that is specifically only for California, we can use a glob pattern of `/California/*` to only use the data in that directory as input to our pipeline.

11.3.3 Only Processing New Data

A datum defines the granularity at which Pachyderm decides what data is new and what data has already been processed. Pachyderm will never reprocess datums it's already seen with the same analysis code. But if any part of a datum changes, the entire datum will be reprocessed.

Note: If you change your code (or pipeline spec), Pachyderm will of course allow you to process all of the past data through the new analysis code.

Let's look at our states example with a few different glob patterns to demonstrate what gets processed and what doesn't. Suppose we have an input data layout such as:

```
/California
   /San-Francisco.json
   /Los-Angeles.json
   ...
/Colorado
   /Denver.json
   /Boulder.json
   ...
...
```

If our glob pattern is `/`, then the entire PFS input is a single datum, which means anytime any file or directory is changed in our input, all the the data will be processed from scratch. There are plenty of usecases where this is exactly what we need (e.g. some machine learning training algorithms).

If our glob pattern is `/*`, then each state directory is its own datum and we'll only process the ones that have changed. So if we add a new city file, `Sacramento.json` to the `/California` directory, *only* the California datum, will be reprocessed.

If our glob pattern was `/*/*` then each `<city>.json` file would be its own datum. That means if we added a `Sacramento.json` file, only that specific file would be processed by Pachyderm.

Incremental Processing

Pachyderm performs computations in an incremental fashion. That is, rather than computing a result all at once, it computes it in small pieces and then stitches those pieces together to form results. This allows Pachyderm to reuse results and compute things much more efficiently than traditional systems, which are forced to compute everything from scratch during every job.

If you are new to the idea of Pachyderm "datums," you can learn more here.

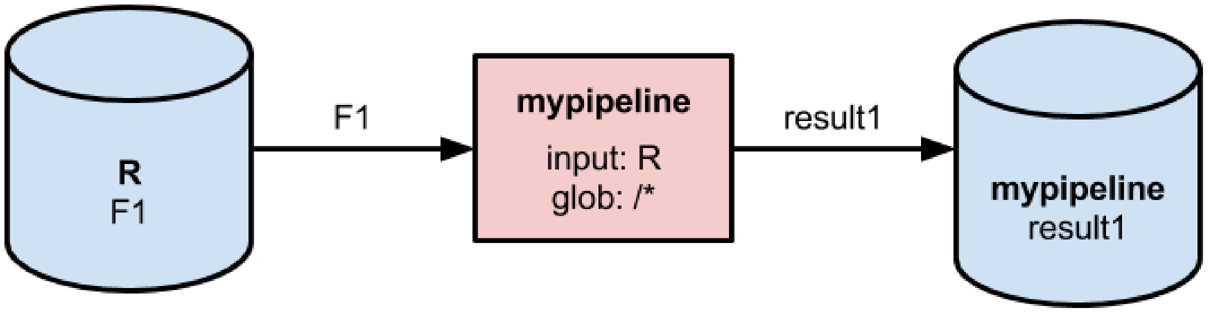
12.1 Inter-datum Incrementality

Each of the input datums in a Pachyderm pipeline is processed in isolation, and the results of these isolated computations are combined to create the final result. Pachyderm will never process the same datum twice (unless you update a pipeline with the `--reprocess` flag). If you commit new data in Pachyderm that leaves some of the previously existing datums intact, the results of processing those pre-existing datums in a previous job will also remain intact. That is, the previous results for those pre-existing datums won't be recalculated.

This inter-datum incrementality is best illustrated with an example. Suppose we have a pipeline with a single input that looks like this:

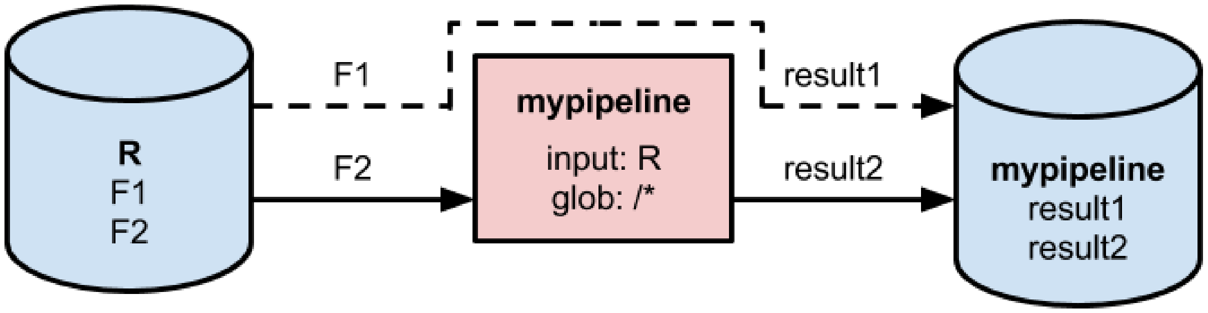
```
{
  "pfs": {
    "repo": "R",
    "glob": "/*",
  }
}
```

Now, suppose you make a commit to `R` which adds a single file `F1`. Your pipeline will run a job, and that job will find a single datum to process (`F1`). This datum will be processed, because it's the first time the pipeline has seen `F1`
.

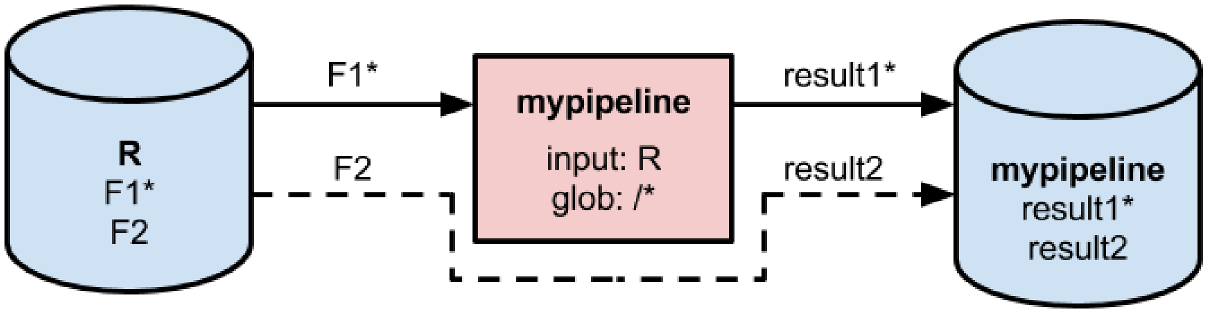


If you then make a second commit to `R` adding another file `F2` , the pipeline will run a second job. This job will find two datums to process (`F1` and `F2`). `F2` will be processed, because it hasn't been seen before. However `F1` will NOT be processed, because an output from processing it already exists in Pachyderm.

Instead, the output from the previous job for `F1` will be combined with the new result from processing `F2` to create the output of this second job. This reuse of the result for `F1` effectively halves the amount of work necessary to process the second commit.



Finally, suppose you make a third commit to `R` , which modifies `F1` . Again you'll have a job that sees two datums (the new `F1` and the already processed `F2`). This time `F2` won't get processed, but the new `F1` will be processed because it has different content as compared to the old `F1` .



Note, you as a user don't need to do anything to enable this inter-datum incrementality. It happens automatically, and it should should be transparent from your perspective. In the above example, you get the same result you would have gotten if you committed the same data in a single commit.

As of Pachyderm v1.5.1, `list job` and `inspect job` will tell you how many datums the job processed and how many it skipped. Below is an example of a job that had 5 datums, 3 that were processed and 2 that were skipped.

```
ID                                   OUTPUT COMMIT                                    ␣
↪STARTED            DURATION            RESTART PROGRESS        DL         UL          STATE
54fbc366-3f11-41f6-9000-60fc8860fa55 pipeline/9c348deb64304d118101e5771e18c2af 13␣
↪seconds ago      10 seconds          0          3 + 2 / 5      0B         0B         success
```

CHAPTER 13

Spouts

13.1 Introduction

Spouts are a way to get streaming data from any source into Pachyderm. To create a spout, you need three things

1. A source of streaming data: a message queue (Kafka, nifi, rabbitMQ, etc.), a database's change feed, the stream of Event Notifications on an S3 bucket (via Amazon SQS), etc.
2. A Docker container holding your spout code that reads from the data source.
3. A spout pipeline specification file that uses the container.

Your spout code will do four things:

1. connect to your source of streaming data
2. read the data
3. package it into files in a `tar` stream
4. write that `tar` stream to the named pipe `/pfs/out`

In this document, we'll take you through writing the spout code (with example code in Golang) and writing the spout pipeline specification.

13.2 Creating your containerized spout code

To create the spout process, you'll need access to client libraries for your streaming data source, a library that can write the `tar` archive format (such as Go's tar package or Python's tarfile module), and requirements for how you would like to batch your data. For the purposes of this document, we'll assume each message in the stream is a single file.

You need that `tar` library because, in spouts, `/pfs/out` is a named pipe. This is different than in pipelines, where `/pfs/out` is a directory.

In the example below, written in Go and taken from the kafka example in the Pachyderm repo, we'll go through every step you need to take.

If you have trouble following this Go code, just read the text to get an idea of what you need to do.

13.2.1 Import necessary libraries

We'll import the libraries necessary for creating a `tar` archive data stream and connecting to our Kafka data source.

```
package main

import (
    "archive/tar"
    "context"
    "os"
    "strings"
    "time"

    kafka "github.com/segmentio/kafka-go"
)
```

13.2.2 Parameterizing connection information

It's a good idea to get parameters for connecting to your data source from the environment or command-line parameters. That way you can connect to new data sources using the same container code without recompiling by just setting appropriate parameters in the pipeline specification.

```
func main() {
    // Get the connection info from the ENV vars
    host := os.Getenv("HOST")
    port := os.Getenv("PORT")
    topic := os.Getenv("TOPIC")
```

13.2.3 Connect to the streaming data source

We're creating an object that can be used to read from our data source.

That `defer` statement is the Go way to guarantee that the open file will be closed after the code in `main()` runs, but before it returns. That is, `reader.Close()` won't be executed until after `main()` is finished with everything else. (This is a common idiom in Go; to `defer` the close of a resource right after you open it.)

```
    // And create a new kafka reader
    reader := kafka.NewReader(kafka.ReaderConfig{
        Brokers:  []string{host + ":" + port},
        Topic:    topic,
        MinBytes: 10e1,
        MaxBytes: 10e6,
    })
    defer reader.Close()
```

13.2.4 Open /pfs/out for writing

We're opening the named pipe `/pfs/out` for writing, so we can send it a `tar` archive stream with the files we want to output. Note that the named pipe has to be opened with write-only permissions.

```
    // Open the /pfs/out pipe with write only permissons (the pachyderm spout will be␣
↪reading at the other end of this)
    // Note: it won't work if you try to open this with read, or read/write␣
↪permissions
    out, err := os.OpenFile("/pfs/out", os.O_WRONLY, 0644)
    if err != nil {
        panic(err)
    }
    defer out.Close()
```

13.2.5 Write the outer file loop

Here we open a tar stream into the directory we opened above, so that Pachyderm can place files into the output repo. For clarity's sake, we'll omit the message-processing loop inside this file loop. The `err` variable is used in the message-processing loop for errors reading from the stream and writing to the directory. The stream is opened at the top of the loop and should be closed at the bottom. In this case, it'll be closed after a message is processed. The commit is finished after the file stream closes. (The `defer tw.Close()` is wrapped in an anonymous function to control when it gets run; it'll run right before the anonymous function is finished.) Think of the `tw.Close()` as a `FinishCommit()`.

```
    // this is the file loop
    for {
        if err := func() error {
            tw := tar.NewWriter(out)
            defer tw.Close()
            // this is the message loop
            for {

                // ...omitted

            }
        }(); err != nil {
            panic(err)
        }
    }
```

13.2.6 Create the message processing loop

Once again, if you have trouble following this Go code, just read the text to get an idea of what you need to do.

First, we read a message from our Kafka queue. Note the use of a 5-second timeout on the read. That's so if the data source, Kafka, hangs for some reason, the spout itself doesn't hang, but gives a hopefully useful error message in the logs after crashing.

```
            // this is the message loop
            for {
                // read a message
                ctx, cancel := context.WithTimeout(context.Background(), 5*time.
↪Second)
                defer cancel()
                m, err := reader.ReadMessage(ctx)
                if err != nil {
                    return err
                }
```

Then we write a filename and the size of the file in a file header to the `tar` stream we opened at the beginning of the file loop. This tar stream will be used by Pachyderm to create files in the output repo.

```
                // give it a unique name
                name := topic + time.Now().Format(time.RFC3339Nano)
                // write the header
                for err = tw.WriteHeader(&tar.Header{
                    Name: name,
                    Mode: 0600,
                    Size: int64(len(m.Value)),
                }); err != nil; {
                    if !strings.Contains(err.Error(), "broken pipe") {
                        return err
                    }
                    // if there's a broken pipe, just give it some time to get ready␣
↪for the next message
                    time.Sleep(5 * time.Millisecond)
                }
```

Then we write the actual message as the contents of the file. Note the use of a timeout in case the named pipe is broken. The reason for this is that the other end of the spout (Pachyderm's code) has closed the named pipe at the end of the previous read. If it gets an error writing to the named pipe, our code should back off, because Pachyderm may not have reopened the named pipe yet.

If you're batching the messages with longer time intervals between writes, this may not be necessary, but it is a good practice to establish for ruggedizing your code. Note: If a more serious error occurs on the named pipe, it may be that a crash has occurred that will be visible in the logs for the pipeline. Any of these errors will be visible in the pipeline's user logs, accessible with `pachctl logs --pipeline=<your pipeline name>`.

```
                // and the message
                for _, err = tw.Write(m.Value); err != nil; {
                    if !strings.Contains(err.Error(), "broken pipe") {
                        return err
                    }
                    // if there's a broken pipe, just give it some time to get ready␣
↪for the next message
                    time.Sleep(5 * time.Millisecond)
                }
                return nil
            }
```

That's the rough outline of operations for processing data in queues and writing it to Pachyderm via a spout.

13.2.7 Create the container for the code

To start, you'll need to create a Dockerfile In our example above, we created a server using Go. The Dockerfile for creating a container with that server in it is in the kafka example.

With your Dockerfile written, you can build your container by running `docker build -t myaccount/myimage:0.1 .` and push it to Docker Hub (or any other registry) with `docker push myaccount/myimage:0.1`.

Once you have containerized your code, you can place it in a Pachyderm spout by writing an appropriate pipeline specification.

13.3 Writing the spout pipeline specification

A spout is defined using a pipeline spec with a `spout` field, and created using the `pachctl create pipeline` command.

Continuing with our example Docker container from above, we might define the specification for the spout as something like this:

```
{
  "pipeline": {
    "name": "my-spout"
  },
  "transform": {
    "cmd": [ "go", "run", "./main.go" ],
    "image": "myaccount/myimage:0.1"
  },
  "env": {
    "HOST": "kafkahost",
    "TOPIC": "mytopic",
    "PORT": "9092"
  },
  "spout": {
    "overwrite": false
  }
}
```

Note that the `overwrite` property on the spout is `false` by default; setting it to `true` would be like having the `--overwrite` flag specified on every `pachctl put file`.

With the spec written, we would then use `pachctl create pipeline -f my-spout.json` to install the spout. It would begin processing messages and placing them in the `my-spout` repo.

CHAPTER 14

History

Pachyderm implements rich version-control and history semantics. This doc lays out the core concepts and architecture of Pachyderm's version-control and the various ways to use the system to access historical data.

14.1 Commits

Commits are the core version-control primitive in Pachyderm, similar to git, commits represent an immutable snapshot of a filesystem and can be accessed with an ID. Commits have a parentage structure, with new commits inheriting content from their parents and then adding to it, you can think of it as a linked list, it's also often referred to as "a chain of commits." Commit IDs are useful if you want to have a static pointer to a snapshot of a filesystem. However, because they're static their use is limited, and you'll mostly deal with branches instead.

14.2 Branches

Branches are pointers to commits, again similar to git, branches have semantically meaningful names such as `master` and `staging`. Branches are mutable, they move along a growing chain of commits as you commit to the branch, and can even be reassigned to any commit within the repo (with `create branch`). The commit that a branch points to is referred to as the branches "head," and the head's ancestors are referred to as "on the branch." Branches can be substituted for commits in Pachyderm's API and will behave as if the head of the branch were passed. This allows you to deal with semantic meaningful names for commits that can be updated, rather than static opaque identifiers.

14.3 Ancestry Syntax

Pachyderm's commits and branches support a familiar git syntax for referencing their history. A commit or branch's parent can be referenced by adding a `^` to the end of the commit or branch. Similar to how `master` resolves to the head commit of `master`, `master^` resolves to the parent of the head commit. You can add multiple `^` s, for example `master^^` resolves to the parent of the parent of the head commit of `master`, and so on. This gets unwieldy quickly so it can also be written as `master^3`, which has the same meaning as `master^^^`. Git supports two characters for ancestor references, `^` and `~` with slightly different meanings, Pachyderm supports both characters as well, for familiarity's sake, but their meaning is identical.

Pachyderm also supports a type of ancestor reference that git doesn't: forward references, these use a different special character `.` and resolve to commits on the beginning of commit chains. For example `master.1` is the first (oldest) commit on the `master` branch, `master.2` is the second commit, and so on.

Note that resolving ancestry syntax requires traversing chains of commits high numbers passed to `^` and low numbers passed to `.` will require traversing a large number of commits which, will take a long time. If you plan to repeatedly access an ancestor it may be worth it to resolve that ancestor to a static commit ID with `inspect commit` and use that ID for future accesses.

14.4 The history flag

Pachyderm also allows you to list the history of objects using a `--history` flag. This flag takes a single argument, an integer, which indicates how many historical versions you want. For example you can get the two most recent versions of a file with the following command:

```
$ pachctl list file repo@master:/file --history 2
COMMIT                           NAME  TYPE COMMITTED      SIZE
73ba56144be94f5bad1ce64e6b96eade /file file 16 seconds ago 8B
c5026f053a7f482fbd719dadecec8f89 /file file 21 seconds ago 4B
```

Note that this isn't necessarily the same result you'd get if you did: `pachctl list file repo@master:/file` followed by `pachctl list file repo@master^:/file`, because the history flag actually looks for changes to the file, and the file need not change in every commit. Similar to the ancestry syntax above, the history flag requires traversing through a linked list of commits, and thus can be expensive. You can get back the full history of a file by passing `all` to the history flag.

14.5 Pipelines

Pipelines are the main processing primitive in Pachyderm, however they expose version-control and history semantics similar to filesystem objects, this is largely because, under the hood, they are implemented in terms of filesystem objects. You can access previous versions of a pipeline using the same ancestry syntax that works for commits and branches, for example `pachctl inspect pipeline foo^` will give you the previous version of the pipeline `foo`, `pachctl inspect pipeline foo.1` will give you the first ever version of that same pipeline. This syntax can be used wherever pipeline names are accepted. A common workflow is reverting a pipeline to a previous version, which can be accomplished with:

```
$ pachctl extract pipeline pipeline^ | pachctl create pipeline
```

Historical versions of pipelines can also be returned with a `--history` flag passed to `pachctl list pipeline` for example:

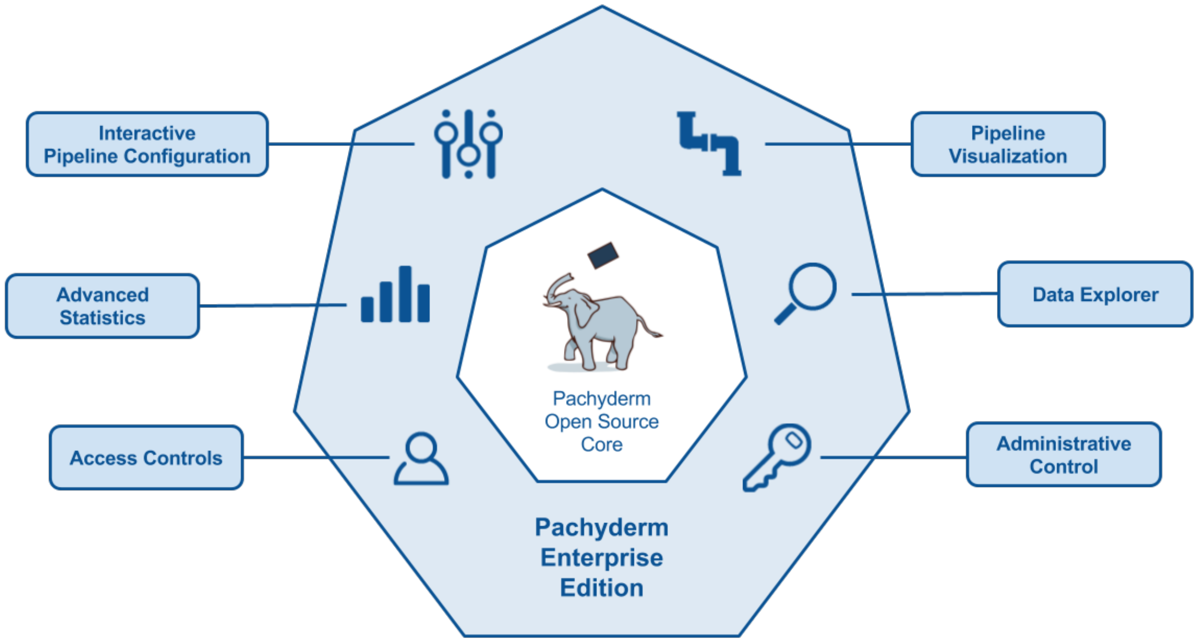
```
$ pachctl list pipeline --history all
NAME      VERSION INPUT     CREATED     STATE / LAST JOB
Pipeline2 1       input2:/* 4 hours ago running / success
Pipeline1 3       input1:/* 4 hours ago running / success
Pipeline1 2       input1:/* 4 hours ago running / success
Pipeline1 1       input1:/* 4 hours ago running / success
```

14.6 Jobs

Jobs do not have versioning semantics associated with them, however, they are associated strongly with the pipelines that created them and thus inherit some of their versioning semantics. This is reflected in `pachctl list job`, by default this command will return all jobs from the most recent versions of all pipelines. You can focus it on a single pipeline by passing `-p pipeline` and you can focus it on a previous version of that pipeline by passing `-p` `pipeline^`. Furthermore you can get jobs from multiple versions of pipelines by passing the `--history` flag, for example: `pachctl list job --history all` will get you all jobs from all versions of all pipelines.

CHAPTER 15

Overview

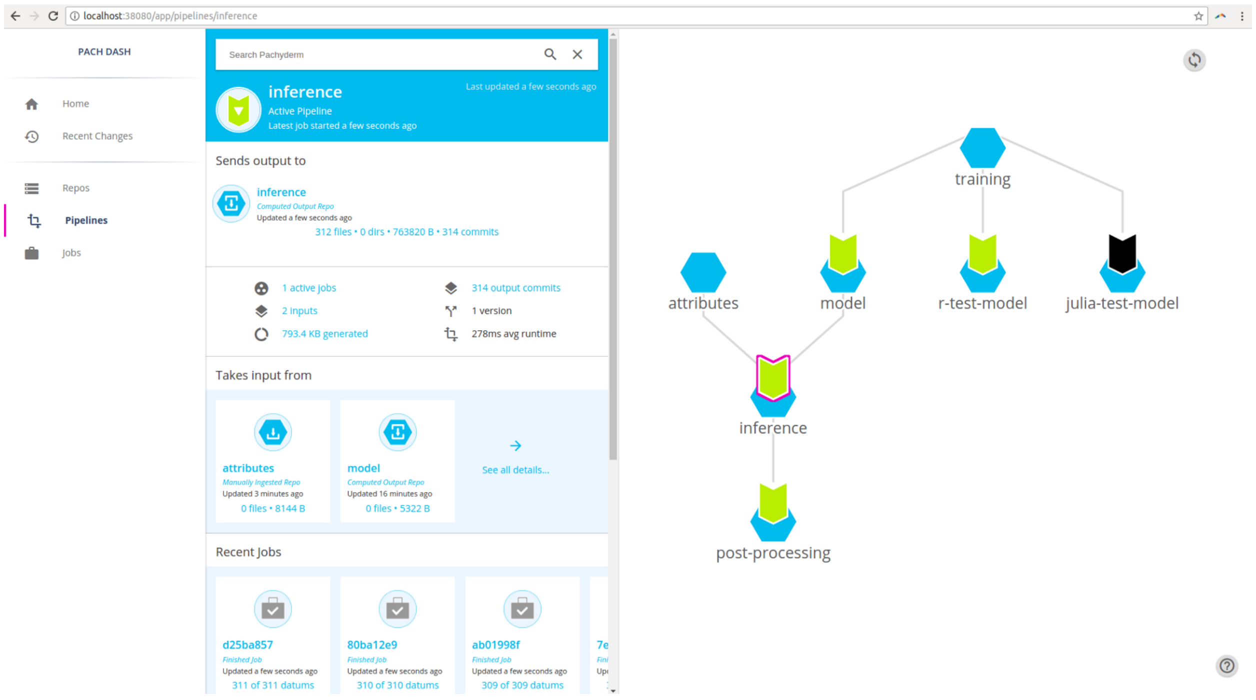


Pachyderm Enterprise Edition includes everything you need to scale and manage Pachyderm data pipelines in an enterprise setting. It delivers the most recent version of Pachyderm along with:

- Administrative and security features needed for enterprise-scale implementations of Pachyderm
- Visual and interactive interfaces to Pachyderm
- Detailed job and data statistics for faster development and data insight

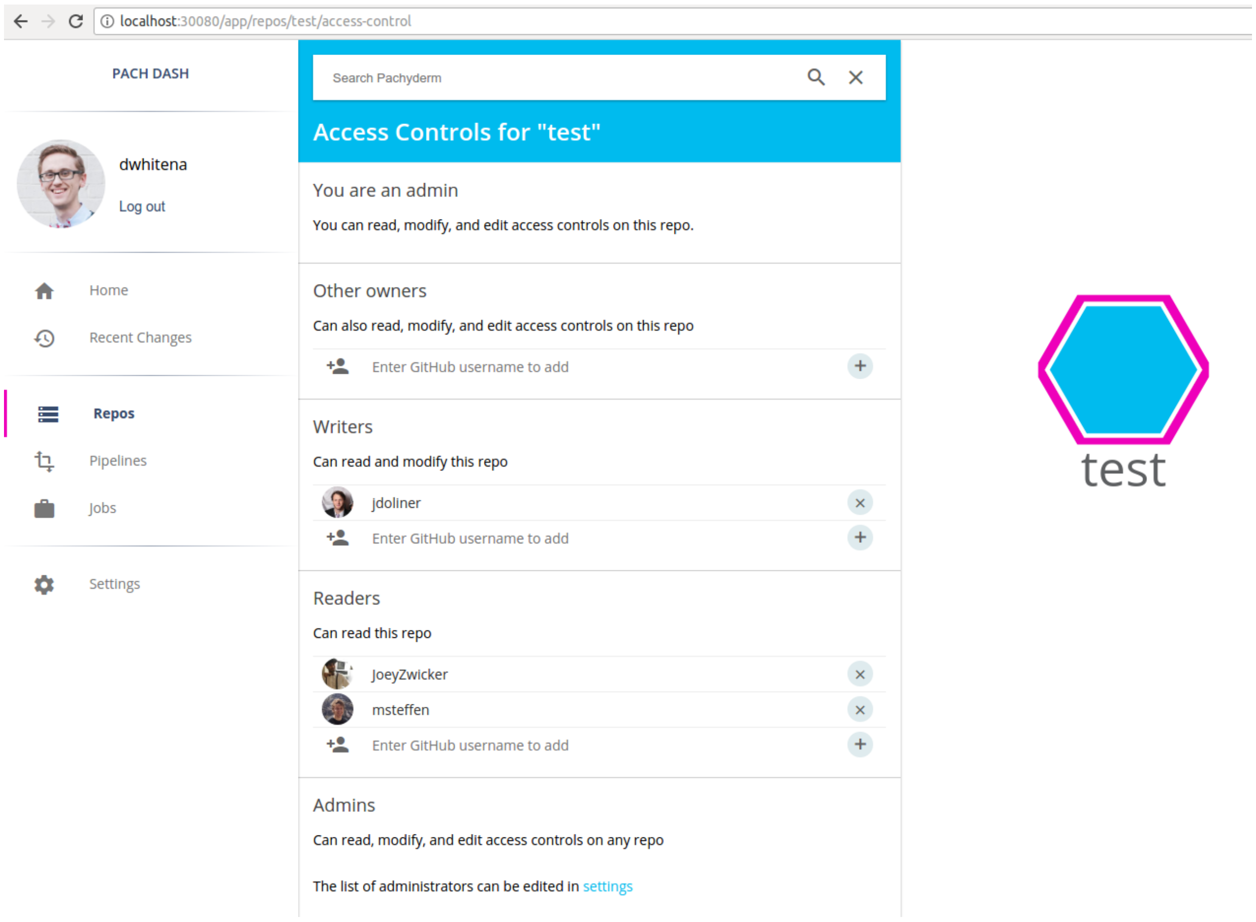
Pachyderm Enterprise Edition can be deployed easily on top of an existing or new deployment of Pachyderm, and we have engineers available to help enterprise customers get up and running very quickly. To get more information about Pachyderm Enterprise Edition, to ask questions, or to get access for evaluation, please contact us at sales@pachyderm.io or on our Slack.

15.1 Pipeline Visualization and Data Exploration



Pachyderm Enterprise Edition includes a full UI for visualizing pipelines and exploring data. Pachyderm Enterprise will automatically infer the structure of data scientists' DAG pipelines and display them visually. Data scientists and cluster admins can even click on individual segments of the pipelines to see what data is being processed, how many jobs have run, what images and commands are being run, and much more! Data scientists can also explore the versioned data in Pachyderm data repositories and see how the state of data has changed over time.

15.2 Access Controls



Access Controls for "test"

You are an admin

You can read, modify, and edit access controls on this repo.

Other owners

Can also read, modify, and edit access controls on this repo

Enter GitHub username to add

Writers

Can read and modify this repo

jdoliner

Enter GitHub username to add

Readers

Can read this repo

JoeyZwicker

msteffen

Enter GitHub username to add

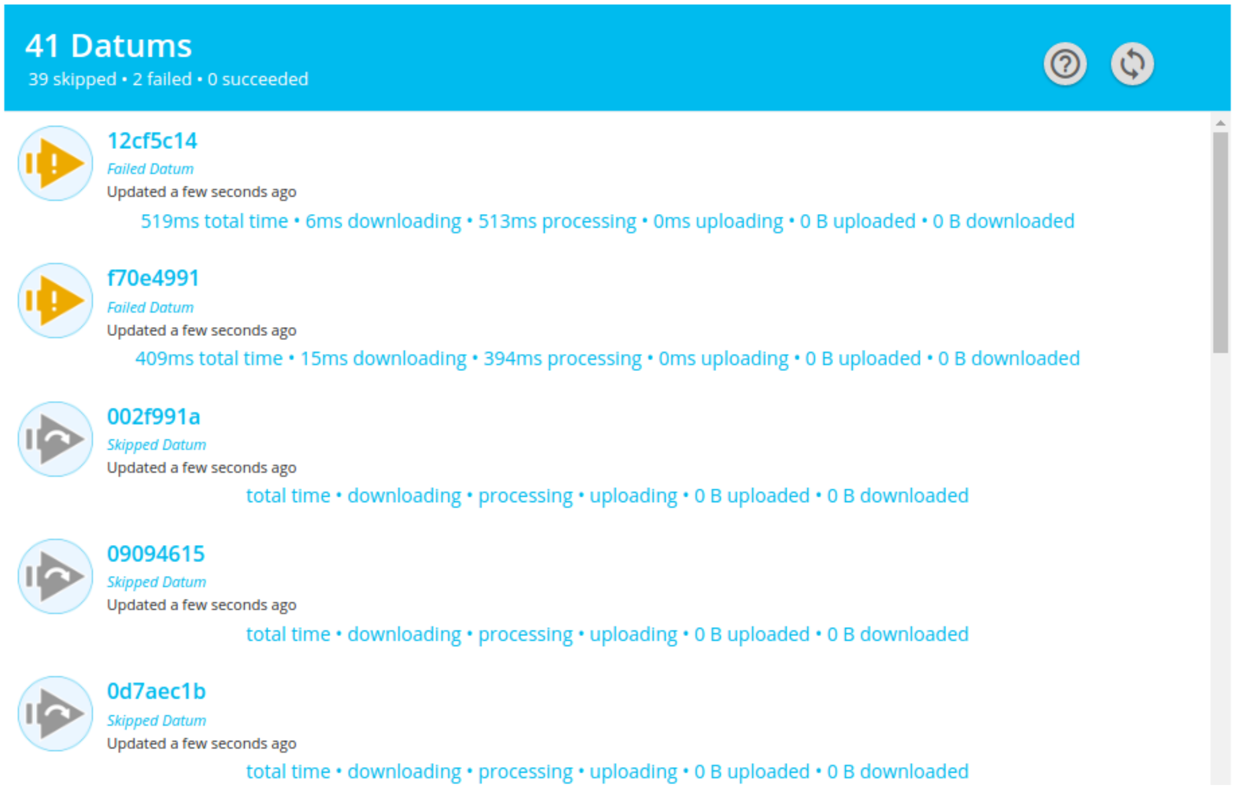
Admins

Can read, modify, and edit access controls on any repo

The list of administrators can be edited in settings

Enterprise-scale deployments require access controls and multitenancy. Pachyderm Enterprise Edition gives teams the ability to control access to production pipelines, data, and configuration. Administrators can silo data, prevent unintended modifications to production pipelines, and support multiple data scientists or even multiple data science groups.

15.3 Advanced Statistics



41 Datums
39 skipped • 2 failed • 0 succeeded

12cf5c14
Failed Datum
Updated a few seconds ago
519ms total time • 6ms downloading • 513ms processing • 0ms uploading • 0 B uploaded • 0 B downloaded

f70e4991
Failed Datum
Updated a few seconds ago
409ms total time • 15ms downloading • 394ms processing • 0ms uploading • 0 B uploaded • 0 B downloaded

002f991a
Skipped Datum
Updated a few seconds ago
total time • downloading • processing • uploading • 0 B uploaded • 0 B downloaded

09094615
Skipped Datum
Updated a few seconds ago
total time • downloading • processing • uploading • 0 B uploaded • 0 B downloaded

0d7aec1b
Skipped Datum
Updated a few seconds ago
total time • downloading • processing • uploading • 0 B uploaded • 0 B downloaded

Pachyderm Enterprise Edition gives data scientists advanced insights into their data, jobs, and results. For example, data scientists can see how much time jobs spend downloading/uploading data, what data was processed or skipped, and which workers were given particular datums. This information can be explored programmatically or via a number of charts and plots that help users parse the information quickly.

15.4 Administrative Controls, Interactive Pipeline Configuration

With Pachyderm Enterprise, cluster admins don't have to rely solely on command line tools and language libraries to configure and control Pachyderm. With new versions of our UI you can control, scale, and configure Pachyderm interactively.

15.5 S3Gateway

Pachyderm Enterprise Edition includes the s3gateway, an S3-like API for interacting with PFS content. With it, you can interact with PFS content with tools and libraries built to work with S3.

Deploying Enterprise Edition

To deploy and use Pachyderm's Enterprise Edition, you simply need to follow one of our guides to deploy Pachyderm and then *activate the Enterprise Edition*.

Note - Pachyderm's Enterprise dashboard is now deployed by default with Pachyderm. If you wish to deploy without the dashboard please use `pachctl deploy [command] --no-dashboard`

Note - You can get a FREE evaluation token for the enterprise edition on the landing page of the Enterprise dashboard.

16.1 Activating Pachyderm Enterprise Edition

There are two ways to activate Pachyderm's enterprise features::

- *Activate Pachyderm Enterprise via the `pachctl` CLI*
- *Activate Pachyderm Enterprise via the dashboard*

For either method, you will need to have your Pachyderm Enterprise activation code available. You should have received this from Pachyderm sales/support when registering for the Enterprise Edition. If you are a new user evaluating Pachyderm, you can receive a FREE evaluation code on the landing page of the dashboard. Please contact support@pachyderm.io if you are having trouble locating your activation code.

16.1.1 Activate via the `pachctl` CLI

Assuming you followed one of our deploy guides and you have a Pachyderm cluster running, you should see that the state of your Pachyderm cluster is similar to the following:

```
$ kubectl get pods
NAME                        READY     STATUS    RESTARTS   AGE
dash-6c9dc97d9c-vb972       2/2       Running   0          6m
etcd-7dbb489f44-9v5jj       1/1       Running   0          6m
pachd-6c878bbc4c-f2h2c      1/1       Running   0          6m
```

You should also be able to connect to the Pachyderm cluster via the `pachctl` CLI:

```
$ pachctl version
COMPONENT           VERSION
pachctl             1.6.8
pachd               1.6.8
```

Activating the Enterprise features of Pachyderm is then as easy as:

```
$ pachctl enterprise activate <activation-code>
```

If this command returns no error, then the activation was successful. The state of the Enterprise activation can also be retrieved at any time via:

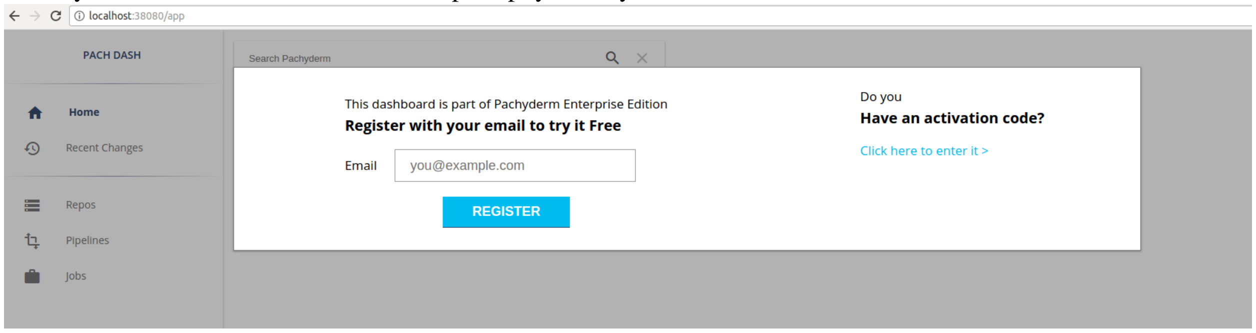
```
$ pachctl enterprise get-state
ACTIVE
```

16.1.2 Activate via the dashboard

You can active Enterprise Edition directly in the dashboard. There's two ways to access the dashboard:

1. If you set the `PACHD_ADDRESS` environment variable, simply point your browser to port 30080 on your kubernetes cluster's IP address.
2. You can enable port forwarding by calling `pachctl port-forward`, then point your browser to `localhost:30080`.

When you first visit the dashboard, it will prompt you for your activation code:



This dashboard is part of Pachyderm Enterprise Edition
Register with your email to try it Free

Do you
Have an activation code?
Click here to enter it >

Once you enter your activation code, you should have full access to the Enterprise dashboard and your cluster will be an active Enterprise Edition cluster. This could be confirmed with:

```
$ pachctl enterprise get-state
ACTIVE
```

CHAPTER 17

Access Controls

The access control features of Pachyderm Enterprise let you create and manage various users that interact with your Pachyderm cluster. You can restrict access to individual data repositories on a per user basis and, as a result, restrict the subscription of pipelines to those data repositories.

These docs will guide you through:

1. *Understanding Pachyderm access controls.*
2. *Activating access control features (aka "auth" features).*
3. *Logging into Pachyderm.*
4. *Managing/updating user access to data repositories.*

We will also discuss:

- *The behavior of pipelines when using access control*
- *The behavior of a cluster when access control is de-activated or an enterprise token expires*

17.1 Understanding Pachyderm access controls

Assuming access controls are activated, each data repository (aka *repo*) in Pachyderm will have an Access Control List (ACL) associated with it. The ACL will include:

- READERs - users who can read the data versioned in the repo.
- WRITERs - users with READER access who can also commit additions, deletions, or modifications of data into the repo.
- OWNERs - users with READER and WRITER access who can also modify the repo's ACL.

Currently, Pachyderm accounts correspond to GitHub users, who authenticate inside of Pachyderm using OAuth integration with GitHub. Pachyderm user accounts are identified within Pachyderm via their GitHub usernames.

There is a single, hardcoded "admin" group (and no other groups) in Pachyderm. Users in that admin group have the ability to perform any action in the cluster, including appointing other admins. Further, a repo with no ACL can only be managed by the cluster admins.

17.2 Activating access control

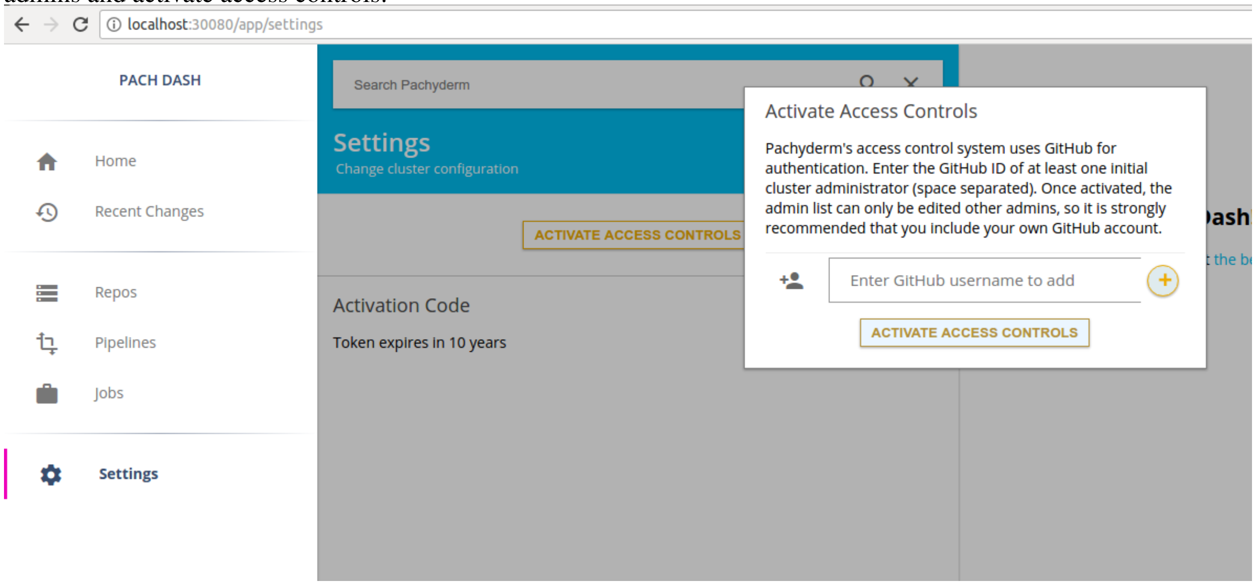
First, you will need to make sure that your cluster has Pachyderm Enterprise Edition activated (you can follow this guide to activate Enterprise Edition). The status of the Enterprise features can be verified by accessing the Pachyderm dashboard or with `pachctl` as follows:

```
$ pachctl enterprise get-state
ACTIVE
```

Next, we need to activate the Enterprise access control features. This can be done *in the dashboard* or with *pachctl auth activate*. However, before executing that command, we should decide on at least one user that will have admin privileges on the cluster. Pachyderm admins will be able to modify the scope of access for any non-admin users on the cluster. All users in Pachyderm are identified by their GitHub usernames.

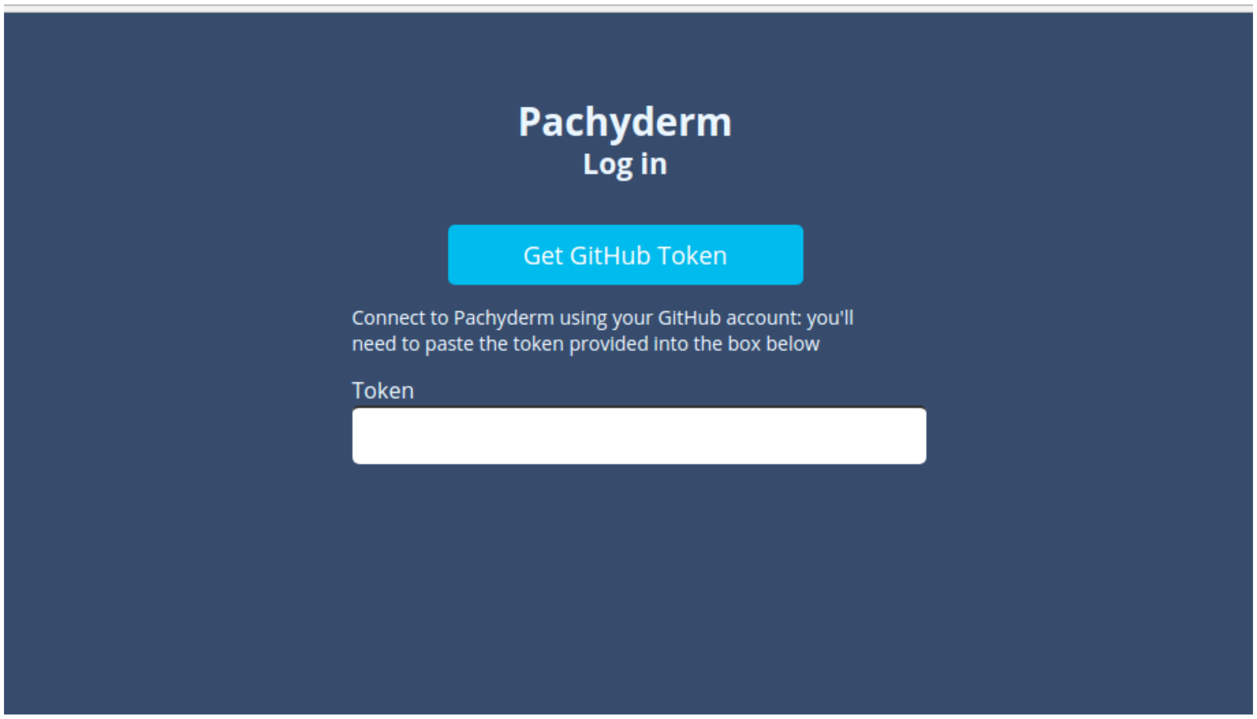
17.2.1 Activating access controls with the dashboard

To activate access controls via the Pachyderm dashboard, go to the settings page where you should see a "Activate Access Controls" button. Click on that button. You will then be able to enter one or more Github users as cluster admins and activate access controls:



Pachyderm's access control system uses GitHub for authentication. Enter the GitHub ID of at least one initial cluster administrator (space separated). Once activated, the admin list can only be edited other admins, so it is strongly recommended that you include your own GitHub account.

After activating access controls, you should see the following screen asking you to login to Pachyderm:



Connect to Pachyderm using your GitHub account: you'll need to paste the token provided into the box below

17.2.2 Activating access controls with pachctl

To activate access controls on a cluster and set the GitHub user `dwhitena` as an admin, we would execute the following `pachctl` command:

```
$ pachctl auth activate --admins=dwhitena
```

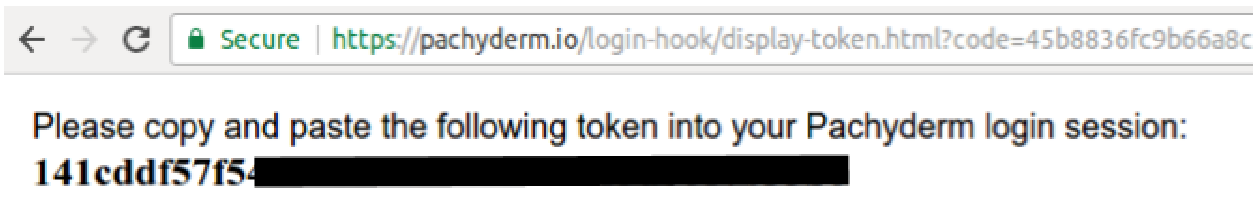
Your Pachyderm cluster can have more than one admin if you like, but you need to supply at least one with this command. To add multiple admins, You would just need to specify them here as a comma separated list.

17.3 Logging into Pachyderm

Now that we have activated access control, we can login to our cluster. When using the Pachyderm dashboard, you will need to *login on the dashboard*, and, when using the `pachctl` CLI, you will need to *login via the CLI*.

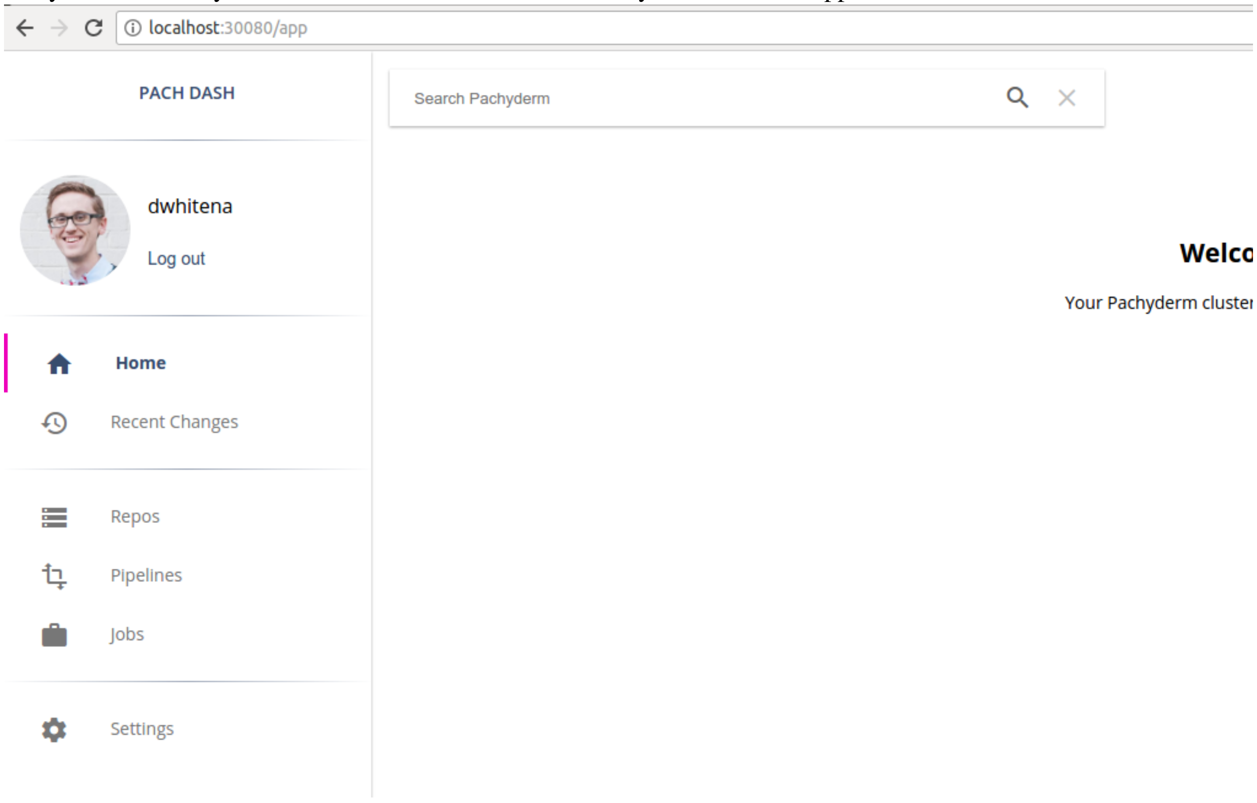
17.3.1 Login on the dashboard

Once you have authorized access controls for Pachyderm, you will need to login to use the Pachyderm dashboard as shown above in *this section*. To login, click the "Get GitHub token" button. You will then be presented with an option to "Authorize Pachyderm" (assuming that you haven't authorized Pachyderm on GitHub previously). Once you authorize Pachyderm, you will be presented with a Pachyderm user token:


Secure | https://pachyderm.io/login-hook/display-token.html?code=45b8836fc9b66a8c

Please copy and paste the following token into your Pachyderm login session:
141cddf57f5[redacted]

Copy and paste this token back into the Pachyderm login screen and press enter. You are now logged in to Pachyderm, and you should see your Github avatar and an indication of your user in the upper left hand corner of the dashboard:

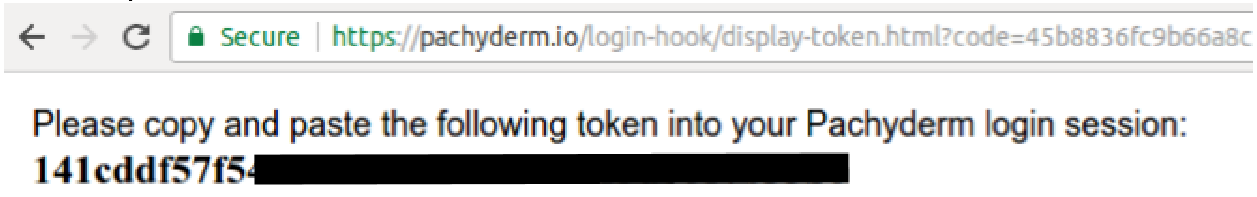


17.3.2 Login using `pachctl`

You can use the `pachctl auth login <username>` to login via the CLI. When we execute this command, `pachctl` will provide us with a GitHub link to authenticate ourselves as the provided GitHub user, as shown below:

```
$ pachctl auth login dwhitena
(1) Please paste this link into a browser:

https://github.com/login/oauth/authorize?client_id=d3481e92b4f09ea74ff8&redirect_
↪uri=https%3A%2F%2Fpachyderm.io%2Flogin-hook%2Fdisplay-token.html

(You will be directed to GitHub and asked to authorize Pachyderm's login app on␣
↪Github. If you accept, you will be given a token to paste here, which will give you␣
↪an externally verified account in this Pachyderm cluster)

(2) Please paste the token you receive from GitHub here:
```

When visiting this link in a browser, you will be presented with an option to "Authorize Pachyderm" (assuming that you haven't authorized Pachyderm via GitHub previously). Once you authorize Pachyderm, you will be presented with a Pachyderm user token:

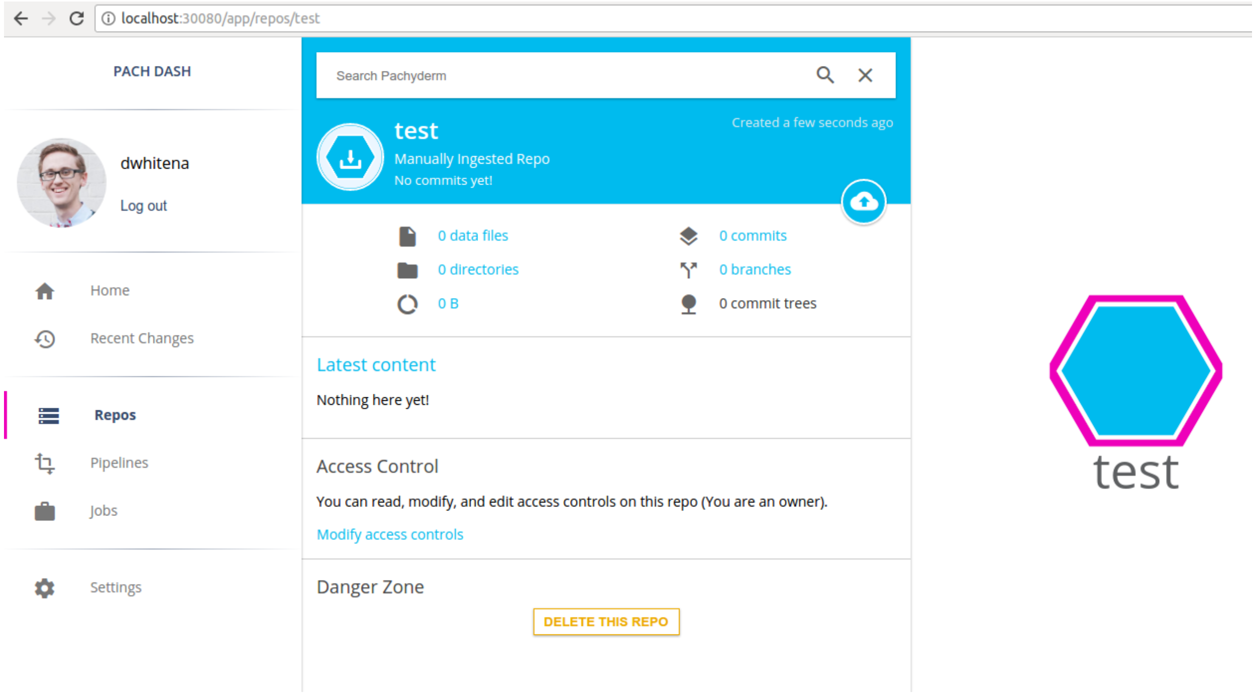

Secure | https://pachyderm.io/login-hook/display-token.html?code=45b8836fc9b66a8c

Please copy and paste the following token into your Pachyderm login session:
141cddf57f54[redacted]

Copy and paste this token back into the terminal, as requested by `pachctl` , and press enter. You are now logged in to Pachyderm!

17.4 Managing and updating user access

Let's suppose that we create a repository call `test` when we are logged into Pachyderm as the user `dwhitena` . Because, the user `dwhitena` created this repository, `dwhitena` will have full read/write access to the repo. This can be confirmed on the dashboard by navigating to or clicking on the repo `test` . The results repo details will show your current access to the repository:


localhost:30080/app/repos/test

PACH DASH

dwhitena
Log out

Home
Recent Changes
Repos
Pipelines
Jobs
Settings

Search Pachyderm

test
Manually Ingested Repo
No commits yet!
Created a few seconds ago

0 data files
0 directories
0 B
0 commits
0 branches
0 commit trees

Latest content
Nothing here yet!

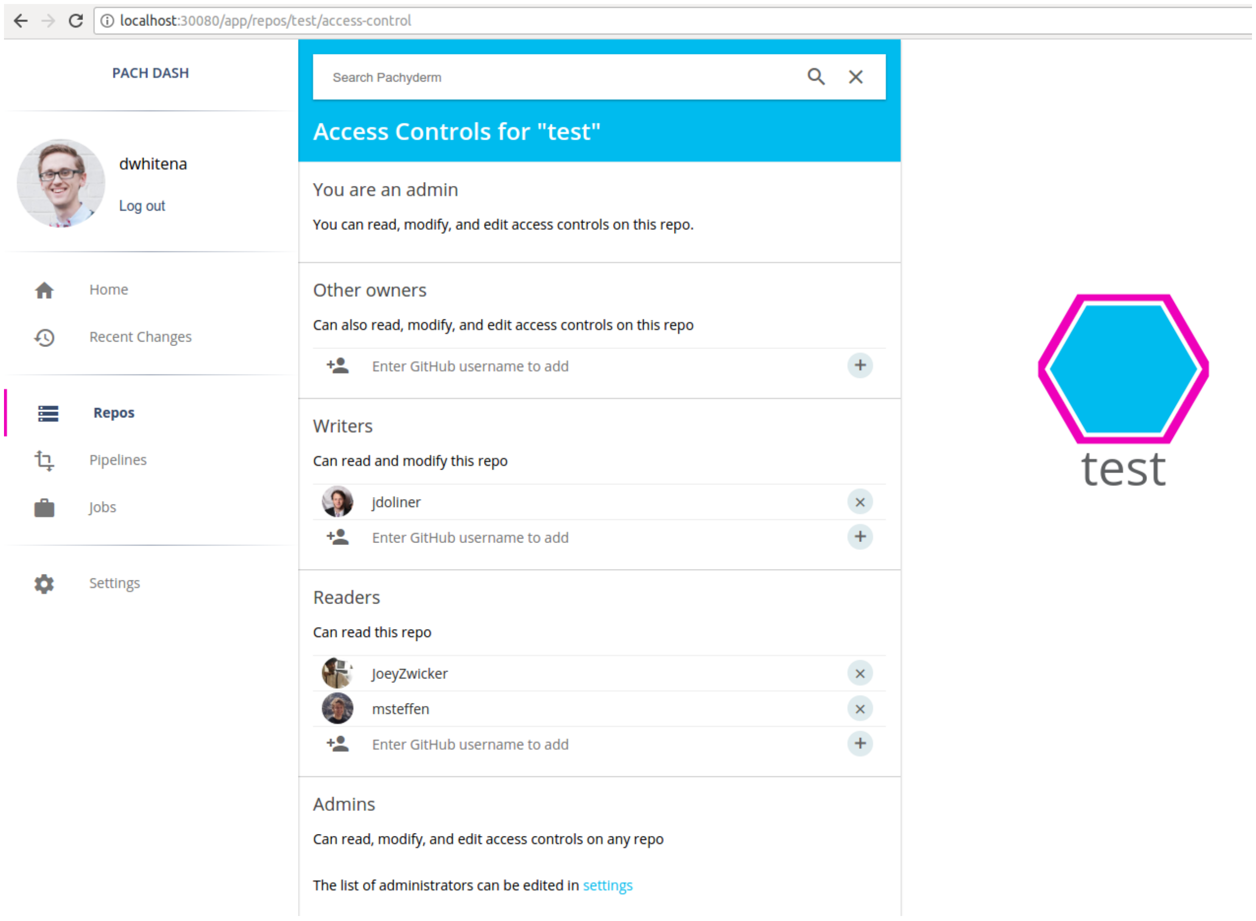
Access Control
You can read, modify, and edit access controls on this repo (You are an owner).
Modify access controls

Danger Zone
DELETE THIS REPO

You can also confirm your access via the `pachctl auth get ...` command:

```
$ pachctl auth get dwhitena test`
OWNER
```

An OWNER of `test` or a cluster admin can then set other user's scope of access to the repo. This can be done via the `pachctl auth set ...` command or via the dashboard. For example, to give the GitHub users `JoeyZwicker` and `msteffen` READER (but not WRITER or OWNER) access to `test` and `jdoliner` WRITER (but not OWNER) access, we can click on `Modify access controls` under the repo details in the dashboard. This will allow us to easily add the users one by one:


localhost:30080/app/repos/test/access-control

PACH DASH

dwhitena

Log out

Home

Recent Changes

Repos

Pipelines

Jobs

Settings

Search Pachyderm

Access Controls for "test"

You are an admin

You can read, modify, and edit access controls on this repo.

Other owners

Can also read, modify, and edit access controls on this repo

Enter GitHub username to add

Writers

Can read and modify this repo

jdoliner

Enter GitHub username to add

Readers

Can read this repo

JoeyZwicker

msteffen

Enter GitHub username to add

Admins

Can read, modify, and edit access controls on any repo

The list of administrators can be edited in settings

17.5 Behavior of pipelines as related to access control

In Pachyderm, you don't explicitly set the scope of access for users on pipelines. Rather, pipelines infer access from the repositories that are input to the pipeline, as follows:

- An OWNER, WRITER, or READER of a repo may subscribe a pipeline to that repo.
- When a user subscribes a pipeline to a repo, they will be set as an OWNER of that pipeline's output repo.
- The initial OWNER of a pipeline's output repo (or an admin) needs to set the scope of access for other users to that output repo.

17.6 Activation code expiration and de-activation

When an enterprise activation code expires, an auth-activated Pachyderm cluster goes into an "admin only" state. In this state, only admins will have access to data that is in Pachyderm. This safety measure keeps sensitive data protected, even when an enterprise subscription becomes stale. As soon as the enterprise activation code is updated (via the dashboard or via `pachctl enterprise activate ...`), the Pachyderm cluster will return to it's previous state.

When access controls are de-activated on a Pachyderm cluster via `pachctl auth deactivate` , the cluster returns to being a non-access controlled Pachyderm cluster. That is,

- All ACLs are deleted.
- The cluster returns to being a blank slate in regards to access control. Everyone that can connect to Pachyderm will be able to access and modify the data in all repos.
- There will no longer be a concept of users (i.e., no one will be able to login to Pachyderm).

Advanced Statistics

To take advantage of the advanced statistics features in Pachyderm Enterprise Edition, you need to:

1. Run your pipelines on a Pachyderm cluster that has activated Enterprise features (see Deploying Enterprise Edition for more details).
2. Enable stats collection in your pipelines by including `"enable_stats": true` in your pipeline specifications.

You will then be able to access the following information for any jobs corresponding to your pipelines:

- The amount of data that was uploaded and downloaded during the job and on a per-datum level (see here for info about Pachyderm datums).
- The time spend uploading and downloading data on a per-datum level.
- The amount of data uploaded and downloaded on a per-datum level.
- The total time spend processing on a per-datum level.
- Success/failure information on a per-datum level.
- The directory structure of input data that was seen by the job.

The primary and recommended way to view this information is via the Pachyderm Enterprise dashboard, which can be deployed as detailed here. However, the same information is available through the `inspect datum` and `list datum pachctl` commands or through their language client equivalents.

Note - We recommend enabling stats for all of your pipeline and only disabling the feature for very stable, long-running pipelines. In most cases, the debugging/maintenance benefits of the stats data will outweigh any disadvantages of storing the extra data associated with the stats. Also note, none of your data is duplicated in producing the stats.

18.1 Enabling stats for a pipeline

As mentioned above, enabling stats collection for a pipeline is as simple as adding the `"enable_stats": true` field to a pipeline specification. For example, to enable stats collection for our OpenCV demo pipeline, we would modify the pipeline specification as follows:

```
{
  "pipeline": {
    "name": "edges"
  },
  "input": {
    "pfs": {
      "glob": "/*",
      "repo": "images"
    }
  },
  "transform": {
    "cmd": [ "python3", "/edges.py" ],
    "image": "pachyderm/opencv"
  },
  "enable_stats": true
}
```

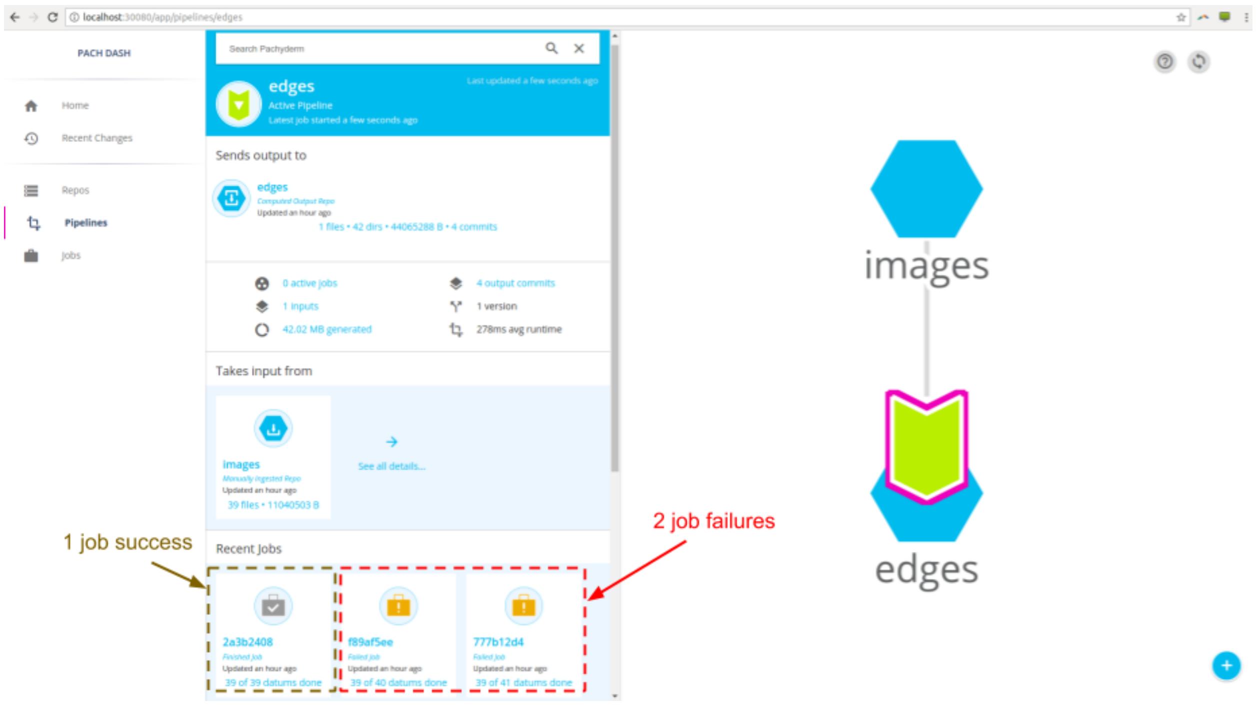
Once the pipeline has been created and you have utilized it to process data, you can confirm that stats are being collected with `list file`. There should now be stats data in the output repo of the pipeline under a branch called `stats`:

```
$ pachctl list file edges@stats
NAME                                                                TYPE                ␣
↪ SIZE
002f991aa9db9f0c44a92a30dff8ab22e788f86cc851bec80d5a74e05ad12868   dir                 ␣
↪ 342.7KiB
0597f2df3f37f1bb5b9bcd6397841f30c62b2b009e79653f9a97f5f13432cf09   dir                 ␣
↪ 1.177MiB
068fac9c3165421b4e54b358630acd2c29f23ebf293e04be5aa52c6750d3374e   dir                 ␣
↪ 270.3KiB
0909461500ce508c330ca643f3103f964a383479097319dbf4954de99f92f9d9   dir                 ␣
↪ 109.6KiB
etc...
```

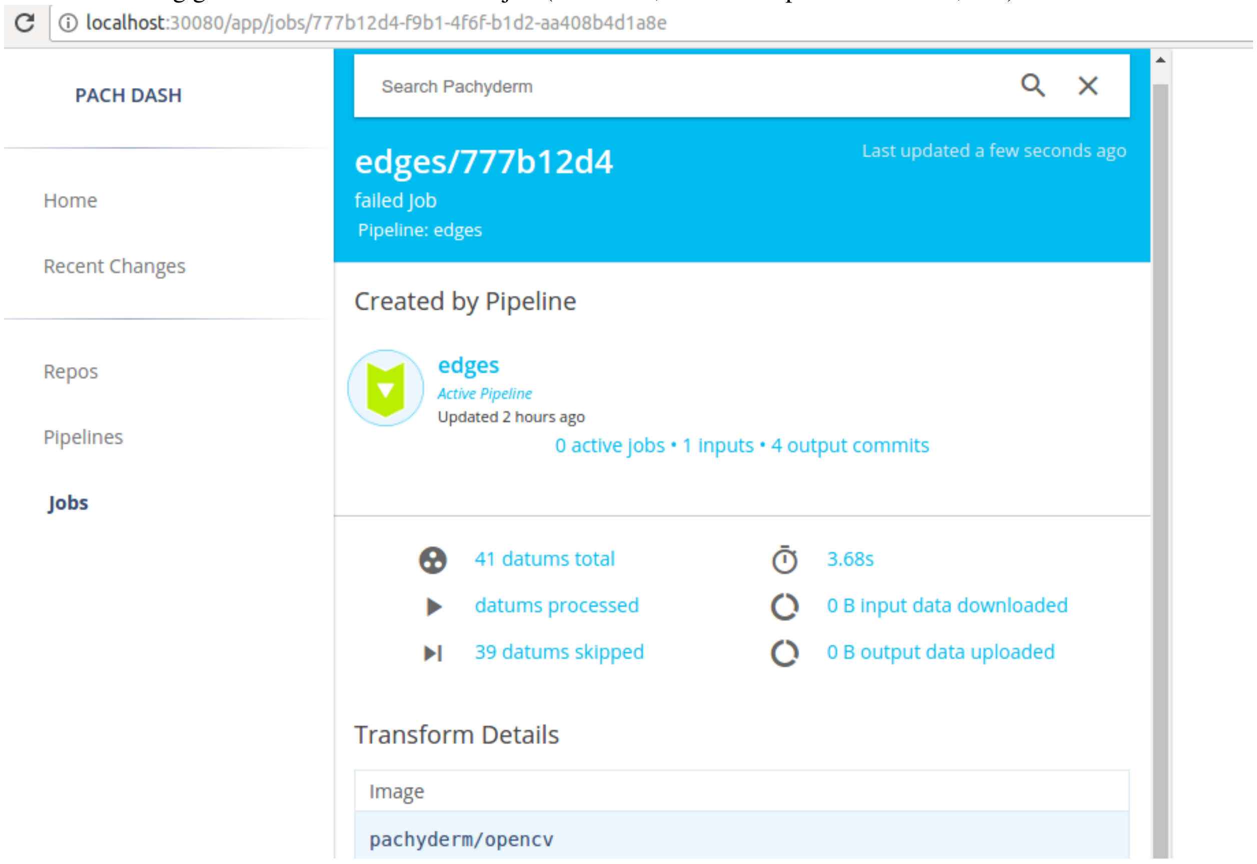
Don't worry too much about this view of the stats data. It just confirms that stats are being collected.

18.2 Accessing stats via the dashboard

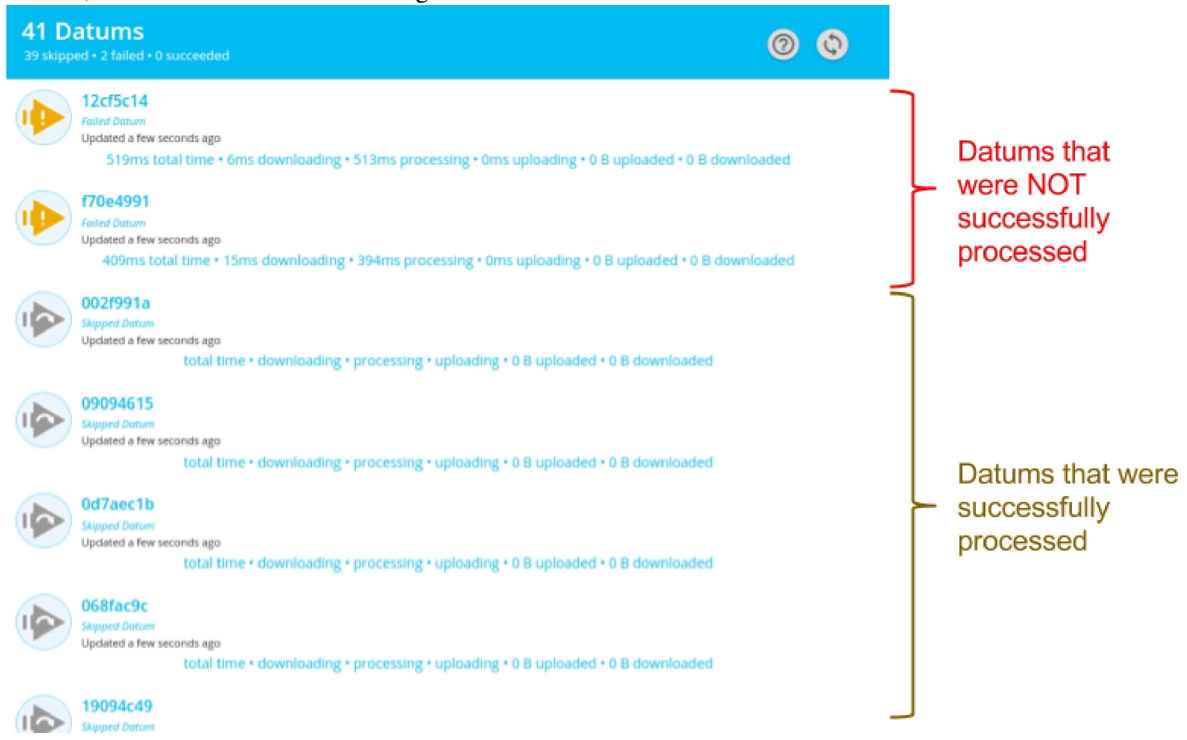
Assuming that you have deployed and activated the Pachyderm Enterprise dashboard, you can explore your advanced statistics in just a few clicks. For example, if we navigate to our `edges` pipeline (specified above), we will see something similar to this:



In this example case, we can see that the pipeline has had 1 recent successful job and 2 recent job failures. Pachyderm advanced stats can be very helpful in debugging these job failures. When we click on one of the job failures we will see the following general stats about the failed job (total time, total data upload/download, etc.):



To get more granular per-datum stats (see here for info on Pachyderm datums), we can click on the `41 datums` `total`, which will reveal the following:


41 Datums
39 skipped • 2 failed • 0 succeeded

12cf5c14
Failed Datum
Updated a few seconds ago
519ms total time • 6ms downloading • 513ms processing • 0ms uploading • 0 B uploaded • 0 B downloaded

f70e4991
Failed Datum
Updated a few seconds ago
409ms total time • 15ms downloading • 394ms processing • 0ms uploading • 0 B uploaded • 0 B downloaded

002f991a
Skipped Datum
Updated a few seconds ago
total time • downloading • processing • uploading • 0 B uploaded • 0 B downloaded

09094615
Skipped Datum
Updated a few seconds ago
total time • downloading • processing • uploading • 0 B uploaded • 0 B downloaded

0d7aec1b
Skipped Datum
Updated a few seconds ago
total time • downloading • processing • uploading • 0 B uploaded • 0 B downloaded

068fac9c
Skipped Datum
Updated a few seconds ago
total time • downloading • processing • uploading • 0 B uploaded • 0 B downloaded

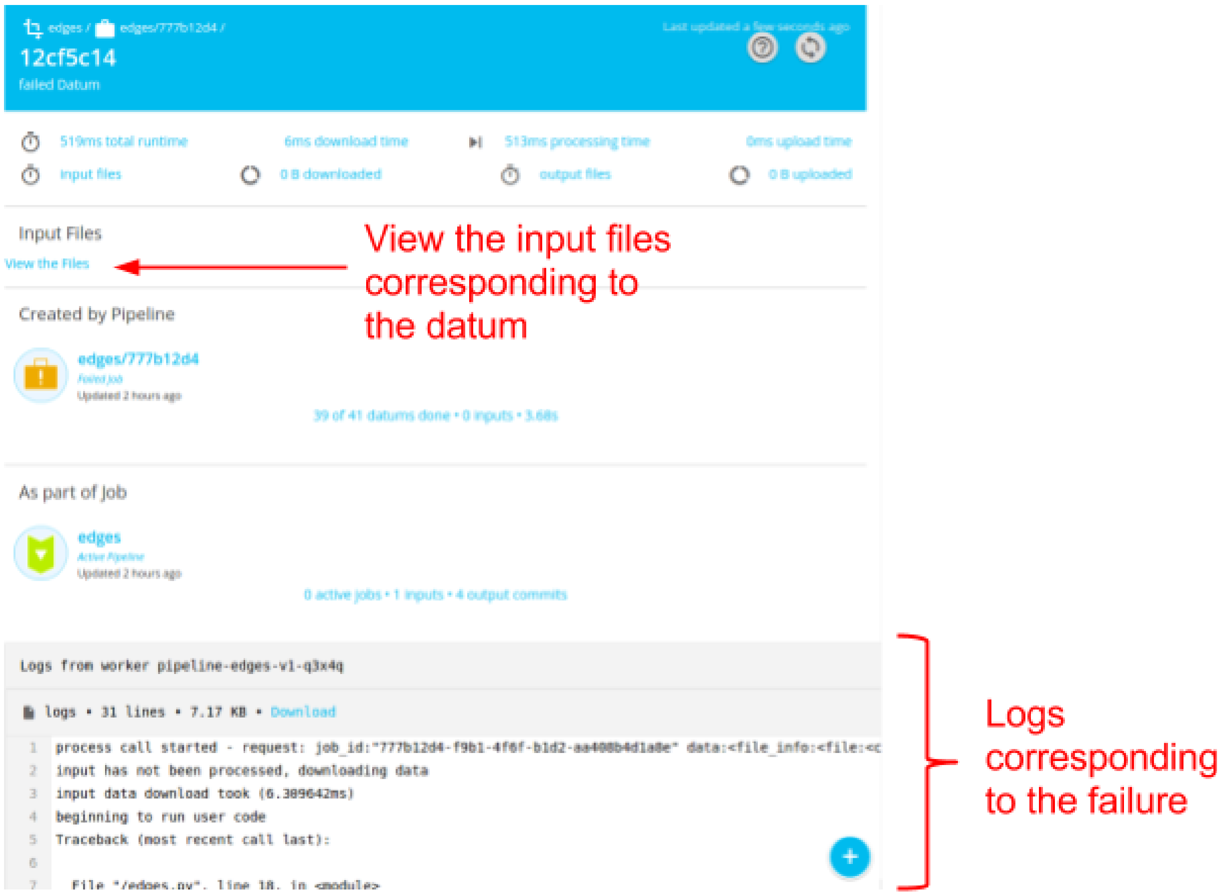
19094c49
Skipped Datum

Datums that were NOT successfully processed
Datums that were successfully processed

We can easily identify the exact datums that caused our pipeline to fail and the associated stats:

- Total time
- Time spent downloading data
- Time spent processing
- Time spent uploading data
- Amount of data downloaded
- Amount of data uploaded

If we need to, we can even go a level deeper and explore the exact details of a failed datum. Clicking on one of the failed datums will reveal the logs corresponding to the datum processing failure along with the exact input files of the datum:



```
Logs from worker pipeline-edges-v1-q3x4q

logs • 31 lines • 7.17 KB • Download

process call started - request: job_id:"777b12d4-f9b1-4f6f-b1d2-aa408b4d1a8e" data:<file_info:<file:<c
input has not been processed, downloading data
input data download took (6.389642ms)
beginning to run user code
Traceback (most recent call last):

  File "/edges.py", line 18, in <module>
```

CHAPTER 19

S3Gateway

Pachyderm Enterprise includes s3gateway, which allows you to interact with PFS storage through an HTTP API that imitates a subset of AWS S3's API. With this, you can reuse a number of tools and libraries built to work with object stores (e.g. minio, Boto) to interact with pachyderm.

You can only interact with the HEAD commit of non-authorization-gated PFS branches through the gateway. If you need richer access, you'll need to work with PFS through its gRPC interface instead.

19.1 Connecting to the s3gateway

The s3gateway runs in your cluster, and can be reached via `http://<cluster ip>:30600`.

Alternatively, you can use port forwarding to connect to the cluster. However, we don't recommend it, as kubernetes' port forwarder incurs overhead, and does not recover well from broken connections.

19.2 Supported operations

These operations are supported by the gateway:

- Creating buckets: creates a repo and/or branch.
- Deleting buckets: Deletings a branch and/or repo.
- Listing buckets: Lists all branches on all repos as s3 buckets.
- Writing objects: Atomically overwrites a file on the HEAD of a branch.
- Removing objects: Atomically removes a file on the HEAD of a branch.
- Listing objects: Lists the files in the HEAD of a branch.
- Getting objects: Gets file contents on the HEAD of a branch.

For details on what's going on under the hood and current peculiarities, see the s3gateway API.

19.3 Unsupported operations

These S3 operations are not supported by the gateway:

- Accelerate

- Analytics
- Object copying. PFS does support this through gRPC though, so if you need this, you can use the gRPC API directly.
- CORS configuration
- Encryption
- HTML form uploads
- Inventory
- Legal holds
- Lifecycles
- Logging
- Metrics
- Multipart uploads. See writing object documentation above for a workaround.
- Notifications
- Object locks
- Payment requests
- Policies
- Public access blocks
- Regions
- Replication
- Retention policies
- Tagging
- Torrents
- Website configuration

Attempting any of these operations will return a standard `NotImplemented` error.

Additionally, there are a few general differences:

- There's no support for authentication or ACLs.
- As per PFS rules, you can't write to an output repo. At the moment, a 500 error will be returned if you try to do so.

19.4 Minio

If you have the option of what S3 client library to use for interfacing with the s3gateway, we recommend minio, as integration with it's go client SDK is thoroughly tested. These minio operations are supported:

Bucket operations

- `MakeBucket`
- `ListBuckets`
- `BucketExists`

- `RemoveBucket`
- `ListObjects`

Object operations

- `GetObject`
- `PutObject`
- `StatObject`
- `RemoveObject`
- `FPutObject`
- `FGetObject`

CHAPTER 20

Overview

Pachyderm runs on Kubernetes and is backed by an object store of your choice. As such, Pachyderm can run on any platform that supports Kubernetes and an object store. These following docs cover common deployments and related topics:

- Google Cloud Platform
- Amazon Web Services
- Azure
- OpenShift
- On Premises
- Custom Object Stores
- Migrations
- Upgrading Pachyderm Versions
- Non-Default Namespaces
- RBAC

20.1 Usage Metrics

Pachyderm automatically reports anonymized usage metrics. These metrics help us understand how people are using Pachyderm and make it better. They can be disabled by setting the env variable `METRICS` to `false` in the pachd container.

Google Cloud Platform

Google Cloud Platform has excellent support for Kubernetes, and thus Pachyderm, through the Google Kubernetes Engine (GKE). The following guide will walk you through deploying a Pachyderm cluster on GCP.

21.1 Prerequisites

- Google Cloud SDK >= 124.0.0
- kubectl
- *pachctl*

If this is the first time you use the SDK, make sure to follow the quick start guide. Note, this may update your `~/.bash_profile` and point your `$PATH` at the location where you extracted `google-cloud-sdk`. We recommend extracting the SDK to `~/bin`.

Note, you can also install `kubectl` installed via the Google Cloud SDK using:

```
$ gcloud components install kubectl
```

21.2 Deploy Kubernetes

To create a new Kubernetes cluster via GKE, run:

```
$ CLUSTER_NAME=<any unique name, e.g. "pach-cluster">

$ GCP_ZONE=<a GCP availability zone. e.g. "us-west1-a">

$ gcloud config set compute/zone ${GCP_ZONE}

$ gcloud config set container/cluster ${CLUSTER_NAME}

$ MACHINE_TYPE=<machine type for the k8s nodes, we recommend "n1-standard-4" or␣
↪larger>

# By default the following command spins up a 3-node cluster. You can change the␣
↪default with `--num-nodes VAL`.
$ gcloud container clusters create ${CLUSTER_NAME} --scopes storage-rw --machine-type
↪${MACHINE_TYPE}
```

```
# By default, GKE clusters have RBAC enabled. To allow 'pachctl deploy' to give the
↪'pachyderm' service account
# the requisite privileges via clusterrolebindings, you will need to grant *your user␣
↪account* the privileges
# needed to create those clusterrolebindings.
#
# Note that this command is simple and concise, but gives your user account more␣
↪privileges than necessary. See
# https://docs.pachyderm.io/en/latest/deployment/rbac.html for the complete list of␣
↪privileges that the
# pachyderm serviceaccount needs.
$ kubectl create clusterrolebinding cluster-admin-binding --clusterrole=cluster-admin␣
↪--user=$(gcloud config get-value account)
```

Important Note: You must create the Kubernetes cluster via the gcloud command-line tool rather than the Google Cloud Console, as it's currently only possible to grant the `storage-rw` scope via the command-line tool. Also note, you should deploy a 1.8.x cluster if possible to take full advantage of Pachyderm's latest features.

This may take a few minutes to start up. You can check the status on the GCP Console. A `kubeconfig` entry will automatically be generated and set as the current context. As a sanity check, make sure your cluster is up and running via `kubectl`:

```
# List all pods in the kube-system namespace.
$ kubectl get pods -n kube-system
NAME                                                      READY     STATUS    RESTARTS␣
↪  AGE
event-exporter-v0.1.7-5c4d9556cf-fd9j2                    2/2       Running   0        ␣
↪  1m
fluentd-gcp-v2.0.9-68vhs                                  2/2       Running   0        ␣
↪  1m
fluentd-gcp-v2.0.9-fzfpw                                  2/2       Running   0        ␣
↪  1m
fluentd-gcp-v2.0.9-qvk8f                                  2/2       Running   0        ␣
↪  1m
heapster-v1.4.3-5fbfb6bf55-xgdwx                          3/3       Running   0        ␣
↪  55s
kube-dns-778977457c-7hbrv                                 3/3       Running   0        ␣
↪  1m
kube-dns-778977457c-dpff4                                 3/3       Running   0        ␣
↪  1m
kube-dns-autoscaler-7db47cb9b7-gp5ns                      1/1       Running   0        ␣
↪  1m
kube-proxy-gke-pach-cluster-default-pool-9762dc84-bzcz   1/1       Running   0        ␣
↪  1m
kube-proxy-gke-pach-cluster-default-pool-9762dc84-hqkr   1/1       Running   0        ␣
↪  1m
kube-proxy-gke-pach-cluster-default-pool-9762dc84-jcbg   1/1       Running   0        ␣
↪  1m
kubernetes-dashboard-768854d6dc-t75rp                     1/1       Running   0        ␣
↪  1m
l7-default-backend-6497bcdb4d-w72k5                       1/1       Running   0        ␣
↪  1m
```

If you *don't* see something similar to the above output, you can point `kubectl` to the new cluster manually via:

```
# Update your kubeconfig to point at your newly created cluster.
$ gcloud container clusters get-credentials ${CLUSTER_NAME}
```

21.3 Deploy Pachyderm

To deploy Pachyderm we will need to:

1. *Create some storage resources*,
2. *Install the Pachyderm CLI tool, `pachctl`*, and
3. *Deploy Pachyderm on the k8s cluster*

21.3.1 Set up the Storage Resources

Pachyderm needs a GCS bucket and a persistent disk to function correctly. We can specify the size of the persistent disk, the bucket name, and create the bucket as follows:

```
# For the persistent disk, 10GB is a good size to start with.
# This stores PFS metadata. For reference, 1GB
# should work fine for 1000 commits on 1000 files.
$ STORAGE_SIZE=<the size of the volume that you are going to create, in GBs. e.g. "10
↪">

# The Pachyderm bucket name needs to be globally unique across the entire GCP region.
$ BUCKET_NAME=<The name of the GCS bucket where your data will be stored>

# Create the bucket.
$ gsutil mb gs://${BUCKET_NAME}
```

To check that everything has been set up correctly, try:

```
$ gsutil ls
# You should see the bucket you created.
```

21.3.2 Install `pachctl`

`pachctl` is a command-line utility for interacting with a Pachyderm cluster. You can install it locally as follows:

```
# For macOS:
$ brew tap pachyderm/tap && brew install pachyderm/tap/pachctl@1.9

# For Linux (64 bit) or Window 10+ on WSL:
$ curl -o /tmp/pachctl.deb -L https://github.com/pachyderm/pachyderm/releases/
↪download/v1.9.0/pachctl_1.9.0_amd64.deb && sudo dpkg -i /tmp/pachctl.deb
```

You can then run `pachctl version --client-only` to check that the installation was successful.

```
$ pachctl version --client-only
1.8.0
```

21.3.3 Deploy Pachyderm on the k8s cluster

Now we're ready to deploy Pachyderm itself. This can be done in one command:

```
$ pachctl deploy google ${BUCKET_NAME} ${STORAGE_SIZE} --dynamic-etcd-nodes=1
serviceaccount "pachyderm" created
storageclass "etcd-storage-class" created
service "etcd-headless" created
statefulset "etcd" created
service "etcd" created
service "pachd" created
deployment "pachd" created
service "dash" created
deployment "dash" created
secret "pachyderm-storage-secret" created

Pachyderm is launching. Check its status with "kubectl get all"
Once launched, access the dashboard by running "pachctl port-forward"
```

Note, here we are using 1 etcd node to manage Pachyderm metadata. The number of etcd nodes can be adjusted as needed.

Important Note: If RBAC authorization is a requirement or you run into any RBAC errors please read our docs on the subject here.

It may take a few minutes for the pachd nodes to be running because it's pulling containers from DockerHub. You can see the cluster status with `kubectl`, which should output the following when Pachyderm is up and running:

```
$ kubectl get pods
NAME                      READY     STATUS    RESTARTS   AGE
dash-482120938-np8cc      2/2       Running   0          4m
etcd-0                    1/1       Running   0          4m
pachd-3677268306-9sqm0    1/1       Running   0          4m
```

If you see a few restarts on the `pachd` pod, that's totally ok. That simply means that Kubernetes tried to bring up those containers before other components were ready, so it restarted them.

Finally, assuming your `pachd` is running as shown above, we need to set up forward a port so that `pachctl` can talk to the cluster.

```
# Forward the ports. We background this process because it blocks.
$ pachctl port-forward &
```

And you're done! You can test to make sure the cluster is working by trying `pachctl version` or even creating a new repo.

```
$ pachctl version
COMPONENT           VERSION
pachctl             1.8.0
pachd               1.8.0
```

21.4 Additional Tips for Performance Improvements

21.4.1 Increasing Ingress Throughput

One way to improve Ingress performance is to restrict Pachd to a specific, more powerful node in the cluster. This is accomplished by the use of node-taints in GKE. By creating a node-taint for Pachd, we're telling the kubernetes scheduler that the only pod that should be on that node is Pachd. Once that's completed, you then deploy Pachyderm with the `--pachd-cpu-request` and `--pachd-memory-request` set to match the resources limits of the machine type. And finally, you'll modify the Pachd deployment such that it has an appropriate toleration:

```
tolerations:
- key: "dedicated"
  operator: "Equal"
  value: "pachd"
  effect: "NoSchedule"
```

21.4.2 Increasing upload performance

The most straightfoward approach to increasing upload performance is to simply leverage SSD's as the boot disk in your cluster as SSD's provide higher throughput and lower latency than standard disks. Additionally, you can increase the size of the SSD for further performance gains as IOPS improve with disk size.

21.4.3 Increasing merge performance

Performance tweaks when it comes to merges can be done directly in the Pachyderm pipeline spec. More specifically, you can increase the number of hashtrees (hashtree spec) in the pipeline spec. This number determines the number of shards for the filesystem metadata. In general this number should be lower than the number of workers (parallelism spec) and should not be increased unless merge time (the time before the job is done and after the number of processed datums + skipped datums is equal to the total datums) is too slow.

CHAPTER 22

Amazon Web Services

22.1 Advanced

- Deploy within an existing VPC
- Connect to your Pachyderm Cluster

22.2 Standard Deployment

We recommend one of the following two methods for deploying Pachyderm on AWS:

1. By manually deploying Kubernetes and Pachyderm.
   - This is appropriate if you (i) already have a kubernetes deployment, (ii) if you would like to customize the types of instances, size of volumes, etc. in your cluster, (iii) if you're setting up a production cluster, or (iv) if you are processing a lot of data or have computationally expensive workloads.
2. By executing a one shot deploy script that will both deploy Kubernetes and Pachyderm.
   - This option is appropriate if you are just experimenting with Pachyderm. The one-shot script will get you up and running in no time!

In addition, we recommend setting up AWS CloudFront for any production deployments. AWS puts S3 rate limits in place that can limit the data throughput for your cluster, and CloudFront helps mitigate this issue. Follow these instructions to deploy with CloudFront

- Deploy a Pachyderm cluster with CloudFront

22.3 Manual Pachyderm Deploy

22.3.1 Prerequisites

- AWS CLI - have it installed and have your AWS credentials configured.
- kubectl
- kops
- *pachctl*

22.3.2 Deploy Kubernetes

The easiest way to install Kubernetes on AWS (currently) is with kops. You can follow this step-by-step guide from Kubernetes for the deploy. Note, we recommend using at `r4.xlarge` or larger instances in the cluster.

Once, you have a Kubernetes cluster up and running in AWS, you should be able to see the following output from `kubectl`:

```
$ kubectl get all
NAME              TYPE        CLUSTER-IP    EXTERNAL-IP    PORT(S)    AGE
svc/kubernetes    ClusterIP   100.64.0.1    <none>         443/TCP    7m
```

22.3.3 Deploy Pachyderm

To deploy Pachyderm on your k8s cluster you will need to:

1. Install the `pachctl` CLI tool,
2. Add some storage resources on AWS,
3. Deploy Pachyderm on top of the storage resources.

Install `pachctl`

To deploy and interact with Pachyderm, you will need `pachctl`, Pachyderm's command-line utility. To install `pachctl` run one of the following:

```
# For macOS:
$ brew tap pachyderm/tap && brew install pachyderm/tap/pachctl@1.9

# For Linux (64 bit) or Window 10+ on WSL:
$ curl -o /tmp/pachctl.deb -L https://github.com/pachyderm/pachyderm/releases/
↪download/v1.9.0/pachctl_1.9.0_amd64.deb && sudo dpkg -i /tmp/pachctl.deb
```

You can try running `pachctl version --client-only` to verify that `pachctl` has been successfully installed.

```
$ pachctl version --client-only
1.8.x
```

Set up the Storage Resources

Pachyderm needs an S3 bucket, and a persistent disk (EBS in AWS) to function correctly.

Here are the environmental variables you should set up to create and utilize these resources:

```
# BUCKET_NAME needs to be globally unique across the entire AWS region
$ BUCKET_NAME=<The name of the S3 bucket where your data will be stored>

# We recommend between 1 and 10 GB. This stores PFS metadata. For reference 1GB
# should work for 1000 commits on 1000 files.
$ STORAGE_SIZE=<the size of the EBS volume that you are going to create, in GBs. e.g.
↪"10">

$ AWS_REGION=<the AWS region of your Kubernetes cluster. e.g. "us-west-2" (not us-
↪west-2a)>
```

Then to actually create the backing S3 bucket, you can run one of the following:

```
# If AWS_REGION is us-east-1.
$ aws s3api create-bucket --bucket ${BUCKET_NAME} --region ${AWS_REGION}

# If AWS_REGION is outside of us-east-1.
$ aws s3api create-bucket --bucket ${BUCKET_NAME} --region ${AWS_REGION} --create-
↪bucket-configuration LocationConstraint=${AWS_REGION}
```

As a sanity check, you should be able to see the bucket that you just created when you run the following:

```
$ aws s3api list-buckets --query 'Buckets[].Name'
```

Deploy Pachyderm

You can deploy Pachyderm on AWS using:

- *An IAM role*, or
- *Static credentials*

Deploying with an IAM role

Run the following command to deploy your Pachyderm cluster:

```
$ pachctl deploy amazon ${BUCKET_NAME} ${AWS_REGION} ${STORAGE_SIZE} --dynamic-etcd-
↪nodes=1 --iam-role <your-iam-role>
```

Note that for this to work, the following need to be true:

- The nodes on which Pachyderm is deployed need to be assigned with the utilized IAM role. If you created your cluster with `kops` , the nodes should have a dedicated IAM role. You can find this IAM role by going to the AWS console, clicking on one of the EC2 instance in the k8s cluster, and inspecting the “Description” of the instance.
- The IAM role needs to have access to the bucket you just created. To ensure that it has access, you can go to the `Permissions` tab of the IAM role and edit the policy to include the following segment (Make sure to replace `your-bucket` with your actual bucket name):

```
{
    "Effect": "Allow",
        "Action": [
            "s3:ListBucket"
        ],
        "Resource": [
            "arn:aws:s3:::your-bucket"
        ]
},
{
    "Effect": "Allow",
    "Action": [
        "s3:PutObject",
    "s3:GetObject",
    "s3:DeleteObject"
    ],
    "Resource": [
        "arn:aws:s3:::your-bucket/*"
    ]
}
```

- The IAM role needs to have the proper "trust relationships" set up. You can verify this by navigating to the `Trust relationships` tab of your IAM role, clicking `Edit trust relationship`, and ensuring that you see a `statement` with `sts:AssumeRole`. For instance, this would be a valid trust relationship:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "ec2.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Once you've run `pachctl deploy ...` and waited a few minutes, you should see the following running pods in Kubernetes:

```
$ kubectl get pods
NAME                        READY     STATUS    RESTARTS   AGE
dash-6c9dc97d9c-89dv9       2/2       Running   0          1m
etcd-0                      1/1       Running   0          4m
pachd-65fd68d6d4-8vjq7      1/1       Running   0          4m
```

Note: If you see a few restarts on the pachd nodes, that's totally ok. That simply means that Kubernetes tried to bring up those containers before etcd was ready so it restarted them.

If you see the above pods running, the last thing you need to do is forward a couple ports so that `pachctl` can talk to the cluster:

```
# Forward the ports. We background this process because it blocks.
$ pachctl port-forward &
```

And you're done! You can verify that the cluster is working by executing `pachctl version`, which should return a version for both `pachctl` and `pachd`:

```
$ pachctl version
COMPONENT           VERSION
pachctl             1.7.0
pachd               1.7.0
```

Deploying with static credentials

When you installed kops, you should have created a dedicated IAM user (see here for details). You could deploy Pachyderm using the credentials of this IAM user directly, although that's not recommended:

```
$ AWS_ACCESS_KEY_ID=<access key ID>

$ AWS_SECRET_ACCESS_KEY=<secret access key>
```

Run the following command to deploy your Pachyderm cluster:

```
$ pachctl deploy amazon ${BUCKET_NAME} ${AWS_REGION} ${STORAGE_SIZE} --dynamic-etcd-
↪nodes=1 --credentials "${AWS_ACCESS_KEY_ID},${AWS_SECRET_ACCESS_KEY},"
```

Note, the `,` at the end of the `credentials` flag in the deploy command is for an optional temporary AWS token. You might utilize this sort of temporary token if you are just experimenting with a deploy. However, such a token should NOT be used for a production deploy.

It may take a few minutes for Pachyderm to start running on the cluster, but you you should eventually see the following running pods:

```
$ kubectl get pods
NAME                     READY     STATUS    RESTARTS   AGE
dash-6c9dc97d9c-89dv9    2/2       Running   0          1m
etcd-0                   1/1       Running   0          4m
pachd-65fd68d6d4-8vjq7   1/1       Running   0          4m
```

If you see an output similar to the above, the last thing you need to do is forward a couple ports so that `pachctl` can talk to the cluster.

```
# Forward the ports. We background this process because it blocks.
$ pachctl port-forward &
```

And you're done! You can verify that the cluster is working by running `pachctl version`, which should return a version for both `pachctl` and `pachd`:

```
$ pachctl version
COMPONENT           VERSION
pachctl             1.7.0
pachd               1.7.0
```

22.4 One Shot Script

22.4.1 Prerequisites

- AWS CLI - have it installed and have your AWS credentials configured.
- kubectl
- kops
- *pachctl*
- jq
- uuid

22.4.2 Run the deploy script

Once you have the prerequisites mentioned above, download and run our AWS deploy script by running:

```
$ curl -o aws.sh https://raw.githubusercontent.com/pachyderm/pachyderm/master/etc/
↪deploy/aws.sh
$ chmod +x aws.sh
$ sudo -E ./aws.sh
```

This script will use `kops` to deploy Kubernetes and Pachyderm in AWS. The script will ask you for your AWS credentials, region preference, etc. If you would like to customize the number of nodes in the cluster, node types, etc., you can open up the deploy script and modify the respective fields.

The script will take a few minutes, and Pachyderm will take an addition couple of minutes to spin up. Once it is up, `kubectl get pods` should return something like:

```
$ kubectl get pods
NAME                        READY     STATUS    RESTARTS   AGE
dash-6c9dc97d9c-89dv9       2/2       Running   0          1m
etcd-0                      1/1       Running   0          4m
pachd-65fd68d6d4-8vjq7      1/1       Running   0          4m
```

22.4.3 Connect `pachctl`

You will then need to forward a couple ports so that `pachctl` can talk to the cluster:

```
# Forward the ports. We background this process because it blocks.
$ pachctl port-forward &
```

And you're done! You can verify that the cluster is working by executing `pachctl version`, which should return a version for both `pachctl` and `pachd`:

```
$ pachctl version
COMPONENT           VERSION
pachctl             1.7.0
pachd               1.7.0
```

22.4.4 Remove

You can delete your Pachyderm cluster using `kops`:

```
$ kops delete cluster
```

In addition, there is the entry in `/etc/hosts` pointing to the cluster that will need to be manually removed. Similarly, kubernetes state s3 bucket and pachyderm storage bucket will need to be manually removed.

CHAPTER 23

Azure

To deploy Pachyderm to Azure, you need to:

1. *Install Prerequisites*
2. *Deploy Kubernetes*
3. *Deploy Pachyderm on Kubernetes*

23.1 Prerequisites

Install the following prerequisites:

- Azure CLI >= 2.0.1
- jq
- kubectl
- *pachctl*

23.2 Deploy Kubernetes

The easiest way to deploy a Kubernetes cluster is through the Azure Container Service (AKS). To create a new AKS Kubernetes cluster using the Azure CLI `az` , run:

```
$ RESOURCE_GROUP=<a unique name for the resource group where Pachyderm will be␣
↪deployed, e.g. "pach-resource-group">

$ LOCATION=<a Azure availability zone where AKS is available, e.g, "Central US">

$ NODE_SIZE=<size for the k8s instances, we recommend at least "Standard_DS4_v2">

$ CLUSTER_NAME=<unique name for the cluster, e.g., "pach-aks-cluster">

# Create the Azure resource group.
$ az group create --name=${RESOURCE_GROUP} --location=${LOCATION}

# Create the AKS cluster.
$ az aks create --resource-group ${RESOURCE_GROUP} --name ${CLUSTER_NAME} --generate-
↪ssh-keys --node-vm-size ${NODE_SIZE}
```

Once Kubernetes is up and running you should be able to confirm the version of the Kubernetes server via:

```
$ kubectl version
Client Version: version.Info{Major:"1", Minor:"9", GitVersion:"v1.9.3", GitCommit:
↪"d2835416544f298c919e2ead3be3d0864b52323b", GitTreeState:"clean", BuildDate:"2018-
↪02-07T12:22:21Z", GoVersion:"go1.9.2", Compiler:"gc", Platform:"darwin/amd64"}
Server Version: version.Info{Major:"1", Minor:"7", GitVersion:"v1.7.9", GitCommit:
↪"19fe91923d584c30bd6db5c5a21e9f0d5f742de8", GitTreeState:"clean", BuildDate:"2017-
↪10-19T16:55:06Z", GoVersion:"go1.8.3", Compiler:"gc", Platform:"linux/amd64"}
```

Note - Azure AKS is still a relatively new managed service. As such, we have had some issues consistently deploying AKS clusters in certain availability zones. If you get timeouts or issues when provisioning an AKS cluster, we recommend trying in a fresh resource group and possibly trying a different zone.

23.3 Deploy Pachyderm

To deploy Pachyderm we will need to:

1. Add some storage resources on Azure,
2. Install the Pachyderm CLI tool, `pachctl` , and
3. Deploy Pachyderm on top of the storage resources.

23.3.1 Set up the Storage Resources

Pachyderm requires an object store and persistent volume (Azure Storage) to function correctly. To create these resources, you need to clone the Pachyderm GitHub repo and then run the following from the root of that repo:

```
$ STORAGE_ACCOUNT=<The name of the storage account where your data will be stored,␣
↪unique in the Azure location>

$ CONTAINER_NAME=<The name of the Azure blob container where your data will be stored>

$ STORAGE_SIZE=<the size of the persistent volume that you are going to create in␣
↪GBs, we recommend at least "10">

# Create an Azure storage account
az storage account create \
  --resource-group="${RESOURCE_GROUP}" \
  --location="${LOCATION}" \
  --sku=Standard_LRS \
  --name="${STORAGE_ACCOUNT}" \
  --kind=Storage

# Build a microsoft tool for creating Azure VMs from an image. Necessary to create␣
↪the blank PV.
$ STORAGE_KEY="$(az storage account keys list \
                 --account-name="${STORAGE_ACCOUNT}" \
                 --resource-group="${RESOURCE_GROUP}" \
                 --output=json \
                 | jq '.[0].value' -r
              )"
```

23.3.2 Install `pachctl`

`pachctl` is a command-line utility used for interacting with a Pachyderm cluster.

```
# For macOS:
$ brew tap pachyderm/tap && brew install pachyderm/tap/pachctl@1.9

# For Linux (64 bit) or Window 10+ on WSL:
$ curl -o /tmp/pachctl.deb -L https://github.com/pachyderm/pachyderm/releases/
↪download/v1.9.0/pachctl_1.9.0_amd64.deb && sudo dpkg -i /tmp/pachctl.deb
```

You can try running `pachctl version` to check that this worked correctly:

```
$ pachctl version --client-only
COMPONENT           VERSION
pachctl             1.7.0
```

23.3.3 Deploy Pachyderm

Now we're ready to deploy Pachyderm:

```
$ pachctl deploy microsoft ${CONTAINER_NAME} ${STORAGE_ACCOUNT} ${STORAGE_KEY} $
↪{STORAGE_SIZE} --dynamic-etcd-nodes 1
```

It may take a few minutes for the pachd pods to be running because it's pulling containers from Docker Hub. When Pachyderm is up and running, you should see something similar to the following state:

```
$ kubectl get pods
NAME                      READY     STATUS    RESTARTS   AGE
dash-482120938-vdlg9      2/2       Running   0          54m
etcd-0                    1/1       Running   0          54m
pachd-1971105989-mjn61    1/1       Running   0          54m
```

Note: If you see a few restarts on the pachd nodes, that's totally ok. That simply means that Kubernetes tried to bring up those containers before etcd was ready so it restarted them.

Finally, assuming you want to connect to the cluster from your local machine (i.e., your laptop), we need to set up forward a port so that `pachctl` can talk to the cluster:

```
# Forward the ports. We background this process because it blocks.
$ pachctl port-forward &
```

And you're done! You can test to make sure the cluster is working by trying `pachctl version` or even by creating a new repo.

```
$ pachctl version
COMPONENT           VERSION
pachctl             1.7.0
pachd               1.7.0
```

CHAPTER 24

OpenShift

OpenShift is a popular enterprise Kubernetes distribution. Pachyderm can run on OpenShift with a few small tweaks in the deployment process, which will be outlined below. Please see *known issues* below for currently issues with OpenShift deployments.

24.1 Prerequisites

Pachyderm needs a few things to install and run successfully in any Kubernetes environment

1. A persistent volume, used by Pachyderm's `etcd` for storage of system metatada. The kind of PV you provision will be dependent on your infrastructure. For example, many on-premises deployments use Network File System (NFS) access to some kind of enterprise storage.
2. An object store, used by Pachyderm's `pachd` for storing all your data. The object store you use will probably be dependent on where you're going to run OpenShift: S3 for AWS, GCS for Google Cloud Platform, Azure Blob Storage for Azure, or a storage provider like Minio, EMC's ECS or Swift providing S3-compatible access to enterprise storage for on-premises deployment.
3. Access to particular TCP/IP ports for communication.

24.1.1 Persistent volume

You'll need to create a persistent volume with enough space for the metadata associated with the data you plan to store Pachyderm. The `pachctl deploy` command for AWS, GCP and Azure creates persistent storage for you, when you follow the instructions below. A custom deploy can also create storage.We'll show you below how to take out the PV that's automatically created, in case you want to create it outside of the Pachyderm deployment and just consume it.

We're currently developing good rules of thumb for scaling this storage as your Pachyderm deployment grows, but it looks like 10G of disk space is sufficient for most purposes.

24.1.2 Object store

Size your object store generously, once you start using Pachyderm, you'll start versioning all your data. You'll need four items to configure object storage

1. The access endpoint. For example, Minio's endpoints are usually something like `minio-server:9000`. Don't begin it with the protocol; it's an endpoint, not an url.
2. The bucket name you're dedicating to Pachyderm. Pachyderm will need exclusive access to this bucket.

3. The access key id for the object store. This is like a user name for logging into the object store.
4. The secret key for the object store. This is like the above user's password.

24.1.3 TCP/IP ports

For more details on how Kubernetes networking and service definitions work, see the Kubernetes services documentation.

Incoming ports (port)

These are the ports internal to the containers, You'll find these on both the pachd and dash containers. OpenShift runs containers and pods as unprivileged users which don't have access to port numbers below 1024. Pachyderm's default manifests use ports below 1024, so you'll have to modify the manifests to use other port numbers. It's usually as easy as adding a "1" in front of the port numbers we use.

Pod ports (targetPort)

This is the port exposed by the pod to Kubernetes, which is forwarded to the `port` . You should leave the `targetPort` set at `0` so it will match the `port` definition.

External ports (nodePorts)

This is the port accessible from outside of Kubernetes. You probably don't need to change `nodePort` values unless your network security requirements or architecture requires you to change to another method of access. Please see the Kubernetes services documentation for details.

24.2 The OCPify script

A bash script that automates many of the substitutions below is available at this gist. You can use it to modify a manifest created using the `--dry-run` flag to `pachctl deploy custom`, as detailed below, and then use this guide to ensure the modifications it makes are relevant to your OpenShift environment. It requires certain prerequisites, just as jq and sponge, found in moreutils.

This script may be useful as a basis for automating redeploys of Pachyderm as needed.

24.2.1 Best practices: Infrastructure as code

We highly encourage you to apply the best practices used in developing software to managing the deployment process.

1. Create scripts that automate as much of your processes as possible and keep them under version control.
2. Keep copies of all artifacts, such as manifests, produced by those scripts and keep those under version control.
3. Document your practices in the code and outside it.

24.3 Preparing to deploy Pachyderm

Things you'll need

1. Your PV. It can be created separately.
2. Your object store information
3. Your project in OpenShift
4. A text editor for editing your deployment manifest

24.4 Deploying Pachyderm

24.4.1 1. Setting up PV and object stores

How you deploy Pachyderm on OpenShift is largely going to depend on where OpenShift is deployed. Below you'll find links to the documentation for each kind of deployment you can do. Follow the instructions there for setting up persistent volumes and object storage resources. Don't yet deploy your manifest, come back here after you've set up your PV and object store.

- OpenShift Deployed on AWS
- OpenShift Deployed on GCP
- OpenShift Deployed on Azure
- OpenShift Deployed on-premise

24.4.2 2. Determine your role security policy

Pachyderm is deployed by default with cluster roles. Many institutional Openshift security policies require namespace-local roles rather than cluster roles. If your security policies require namespace-local roles, use the *`pachctl deploy` command below with the `--local-roles` flag*.

24.4.3 3. Run the deploy command with –dry-run

Once you have your PV, object store, and project, you can create a manifest for editing using the `--dry-run` argument to `pachctl deploy`. That step is detailed in the deployment instructions for each type of deployment, above.

Below, find examples, with cluster roles and with namespace-local roles, using AWS elastic block storage as a persistent disk with a custom deploy. We'll show how to remove this PV in case you want to use a PV you create separately.

Cluster roles

```
$ pachctl deploy custom --persistent-disk aws --object-store s3 \
     <pv-storage-name> <pv-storage-size> \
     <s3-bucket-name> <s3-access-key-id> <s3-access-secret-key> <s3-access-endpoint-
↪url> \
     --static-etcd-volume=<pv-storage-name> > manifest.json
```

Namespace-local roles

```
$ pachctl deploy custom --persistent-disk aws --object-store s3 \
     <pv-storage-name> <pv-storage-size> \
     <s3-bucket-name> <s3-access-key-id> <s3-access-secret-key> <s3-access-endpoint-
↪url> \
     --static-etcd-volume=<pv-storage-name> --local-roles > manifest.json
```

24.4.4 4. Modify pachd Service ports

In the deployment manifest, which we called `manifest.json`, above, find the stanza for the `pachd` Service. An example is shown below.

```
{
    "kind": "Service",
    "apiVersion": "v1",
    "metadata": {
        "name": "pachd",
        "namespace": "default",
        "creationTimestamp": null,
        "labels": {
            "app": "pachd",
            "suite": "pachyderm"
        },
        "annotations": {
            "prometheus.io/port": "9091",
            "prometheus.io/scrape": "true"
        }
    },
    "spec": {
        "ports": [
            {
                "name": "api-grpc-port",
                "port": 650,
                "targetPort": 0,
                "nodePort": 30650
            },
            {
                "name": "trace-port",
                "port": 651,
                "targetPort": 0,
                "nodePort": 30651
            },
            {
                "name": "api-http-port",
                "port": 652,
                "targetPort": 0,
                "nodePort": 30652
            },
            {
                "name": "saml-port",
                "port": 654,
                "targetPort": 0,
                "nodePort": 30654
            },
            {
                "name": "api-git-port",
                "port": 999,
                "targetPort": 0,
                "nodePort": 30999
            },
            {
                "name": "s3gateway-port",
                "port": 600,
                "targetPort": 0,
                "nodePort": 30600
            }
        ],
        "selector": {
            "app": "pachd"
        },
        "type": "NodePort"
    },
    "status": {
        "loadBalancer": {}
    }
}
```

While the nodePort declarations are fine, the port declarations are too low for OpenShift. Good example values are shown below.

```
    "spec": {
        "ports": [
            {
                "name": "api-grpc-port",
                "port": 1650,
                "targetPort": 0,
                "nodePort": 30650
            },
            {
                "name": "trace-port",
                "port": 1651,
                "targetPort": 0,
                "nodePort": 30651
            },
            {
                "name": "api-http-port",
                "port": 1652,
                "targetPort": 0,
                "nodePort": 30652
            },
            {
                "name": "saml-port",
                "port": 1654,
                "targetPort": 0,
                "nodePort": 30654
            },
            {
                "name": "api-git-port",
                "port": 1999,
                "targetPort": 0,
                "nodePort": 30999
            },
            {
                "name": "s3gateway-port",
                "port": 1600,
                "targetPort": 0,
                "nodePort": 30600
            }
        ],
```

24.4.5 5. Modify pachd Deployment ports and add environment variables

In this case you're editing two parts of the `pachd` Deployment json.Here, we'll omit the example of the unmodified version. Instead, we'll show you the modified version.

5.1 pachd Deployment ports

The `pachd` Deployment also has a set of port numbers in the spec for the `pachd` container. Those must be modified to match the port numbers you set above for each port.

```
{
    "kind": "Deployment",
    "apiVersion": "apps/v1beta1",
    "metadata": {
        "name": "pachd",
        "namespace": "default",
        "creationTimestamp": null,
        "labels": {
            "app": "pachd",
            "suite": "pachyderm"
        }
    },
    "spec": {
        "replicas": 1,
        "selector": {
            "matchLabels": {
                "app": "pachd",
                "suite": "pachyderm"
            }
        },
        "template": {
            "metadata": {
                "name": "pachd",
                "namespace": "default",
                "creationTimestamp": null,
                "labels": {
                    "app": "pachd",
                    "suite": "pachyderm"
                },
                "annotations": {
                    "iam.amazonaws.com/role": ""
                }
            },
            "spec": {
                "volumes": [
                    {
                        "name": "pach-disk"
                    },
                    {
                        "name": "pachyderm-storage-secret",
                        "secret": {
                            "secretName": "pachyderm-storage-secret"
                        }
                    }
                ],
                "containers": [
                    {
                        "name": "pachd",
                        "image": "pachyderm/pachd:1.9.0rc1",
                        "ports": [
                            {
                                "name": "api-grpc-port",
                                "containerPort": 1650,
                                "protocol": "TCP"
                            },
                            {
                                "name": "trace-port",
                                "containerPort": 1651
                            },
                            {
                                "name": "api-http-port",
                                "containerPort": 1652,
                                "protocol": "TCP"
                            },
                            {
                                "name": "peer-port",
                                "containerPort": 1653,
                                "protocol": "TCP"
                            },
                            {
                                "name": "api-git-port",
                                "containerPort": 1999,
                                "protocol": "TCP"
                            },
                            {
                                "name": "saml-port",
                                "containerPort": 1654,
                                "protocol": "TCP"
                            }
                        ],
```

5.2 Add environment variables

There are six environment variables necessary for OpenShift

1. `WORKER_USES_ROOT` : This controls whether worker pipelines run as the root user or not. You'll need to set it to `false`
2. `PORT` : This is the grpc port used by pachd for communication with `pachctl` and the api. It should be set to the same value you set for `api-grpc-port` above.
3. `PPROF_PORT` : This is used for Prometheus. It should be set to the same value as `trace-port` above.
4. `HTTP_PORT` : The port for the api proxy. It should be set to `api-http-port` above.

5. PEER_PORT : Used to coordinate pachd 's. Same as peer-port above.
6. PPS_WORKER_GRPC_PORT : Used to talk to pipelines. Should be set to a value above 1024. The example value of 1680 below is recommended.

The added values below are shown inserted above the PACH_ROOT value, which is typically the first value in this array. The rest of the stanza is omitted for clarity.

```
                        "env": [
                            {
                            "name": "WORKER_USES_ROOT",
                            "value": "false"
                            },
                            {
                            "name": "PORT",
                            "value": "1650"
                            },
                            {
                            "name": "PPROF_PORT",
                            "value": "1651"
                            },
                            {
                            "name": "HTTP_PORT",
                            "value": "1652"
                            },
                            {
                            "name": "PEER_PORT",
                            "value": "1653"
                            },
                            {
                            "name": "PPS_WORKER_GRPC_PORT",
                            "value": "1680"
                            },
                            {
                                "name": "PACH_ROOT",
                                "value": "/pach"
                            },
```

24.4.6 6. (Optional) Remove the PV created during the deploy command

If you're using a PV you've created separately, remove the PV that was added to your manifest by pachctl deploy --dry-run . Here's the example PV we created with the deploy command we used above, so you can recognize it.

```
{
    "kind": "PersistentVolume",
    "apiVersion": "v1",
    "metadata": {
        "name": "etcd-volume",
        "namespace": "default",
        "creationTimestamp": null,
        "labels": {
            "app": "etcd",
            "suite": "pachyderm"
        }
    },
    "spec": {
        "capacity": {
            "storage": "10Gi"
        },
        "awsElasticBlockStore": {
            "volumeID": "pach-disk",
            "fsType": "ext4"
        },
        "accessModes": [
            "ReadWriteOnce"
        ],
        "persistentVolumeReclaimPolicy": "Retain"
    },
    "status": {}
}
```

24.5 7. Deploy the Pachyderm manifest you modified.

```
$ oc create -f pachyderm.json
```

You can see the cluster status by using `oc get pods` as in upstream Kubernetes:

```
    $ oc get pods
    NAME                        READY     STATUS    RESTARTS   AGE
    dash-6c9dc97d9c-89dv9       2/2       Running   0          1m
    etcd-0                      1/1       Running   0          4m
    pachd-65fd68d6d4-8vjq7      1/1       Running   0          4m
```

24.5.1 Known issues

Problems related to OpenShift deployment are tracked in issues with the "openshift" label.

CHAPTER 25

On Premises

This document is broken down into the following sections, available at the links below

- *Introduction to on-premises deployments* takes you through what you need to know about Kubernetes, persistent volumes, object stores and best practices. That's this page.
- Customizing your Pachyderm deployment for on-premises use details the various options of the `pachctl deploy custom ...` command for an on-premises deployment.
- Single-node Pachyderm deployment is the document you should read when deploying Pachyderm for personal, low-volume usage.
- Registries takes you through on-premises, private Docker registry configuration.
- Ingress details the Kubernetes ingress configuration you'd need for using `pachctl` and the dashboard outside of the Kubernetes cluster
- Non-cloud object stores discusses common configurations for on-premises object stores.

Need information on a particular flavor of Kubernetes or object store? Check out the *see also* section.

Troubleshooting a deployment? Check out Troubleshooting Deployments.

25.1 Introduction

Deploying Pachyderm successfully on-premises requires a few prerequisites and some planning. Pachyderm is built on Kubernetes. Before you can deploy Pachyderm, you or your Kubernetes administrator will need to perform the following actions:

1. *Deploy Kubernetes* on-premises.
2. *Deploy a Kubernetes persistent volume* that Pachyderm will use to store administrative data.
3. *Deploy an on-premises object store* using a storage provider like MinIO, EMC's ECS, or SwiftStack to provide S3-compatible access to your on-premises storage.
4. Create a Pachyderm manifest by running the `pachctl deploy custom` command with appropriate arguments and the `--dry-run` flag to create a Kubernetes manifest for the Pachyderm deployment.
5. *Edit the Pachyderm manifest* for your particular Kubernetes deployment

In this series of documents, we'll take you through the steps unique to Pachyderm. We assume you have some Kubernetes knowledge. We will point you to external resources for the general Kubernetes steps to give you background.

25.2 Best practices

25.2.1 Infrastructure as code

We highly encourage you to apply the best practices used in developing software to managing the deployment process.

1. Create scripts that automate as much of your processes as possible and keep them under version control.
2. Keep copies of all artifacts, such as manifests, produced by those scripts and keep those under version control.
3. Document your practices in the code and outside it.

25.2.2 Infrastructure in general

Be sure that you design your Kubernetes infrastructure in accordance with recommended guidelines. Don't mix on-premises Kubernetes and cloud-based storage. It's important that bandwidth to your storage deployment meet the guidelines of your storage provider.

25.3 Prerequisites

25.3.1 Software you will need

1. kubectl
2. pachctl

25.4 Setting up to deploy on-premises

25.4.1 Deploying Kubernetes

The Kubernetes docs have instructions for deploying Kubernetes in a variety of on-premise scenarios. We recommend following one of these guides to get Kubernetes running on premise.

25.4.2 Deploying a persistent volume

Persistent volumes: how do they work?

A Kubernetes persistent volume is used by Pachyderm's `etcd` for storage of system metatada. In Kubernetes, persistent volumes are a mechanism for providing storage for consumption by the users of the cluster. They are provisioned by the cluster administrators. In a typical enterprise Kubernetes deployment, the administrators have configured persistent volumes that your Pachyderm deployment will consume by means of a persistent volume claim in the Pachyderm manifest you generate.

You can deploy PV's to Pachyderm using our command-line arguments in three ways: using a static PV, with StatefulSets, or with StatefulSets using a StorageClass.

If your administrators are using selectors, or you want to use StorageClasses in a different way, you'll need to *edit the Pachyderm manifest* appropriately before applying it.

Static PV

In this case, `etcd` will be deployed in Pachyderm as a ReplicationController with one (1) pod that uses a static PV. This is a common deployment for testing.

StatefulSets

StatefulSets are a mechanism provided in Kubernetes 1.9 and newer to manage the deployment and scaling of applications. It uses either Persistent Volume Provisioning or pre-provisioned PV's.

If you're using StatefulSets in your Kubernetes cluster, you will need to find out the particulars of your cluster's PV configuration and *use appropriate flags to* `pachctl deploy custom`

StorageClasses

If your administrators require specification of classes to consume persistent volumes, you will need to find out the particulars of your cluster's PV configuration and *use appropriate flags to* `pachctl deploy custom`.

Common tasks to all types of PV deployments

Sizing the PV

You'll need to use a PV with enough space for the metadata associated with the data you plan to store in Pachyderm. We're currently developing good rules of thumb for scaling this storage as your Pachyderm deployment grows, but it looks like 10G of disk space is sufficient for most purposes.

Creating the PV

In the case of cloud-based deployments, the `pachctl deploy` command for AWS, GCP and Azure creates persistent volumes for you, when you follow the instructions for those infrastructures.

In the case of on-premises deployments, the kind of PV you provision will be dependent on what kind of storage your Kubernetes administrators have attached to your cluster and configured, and whether you are expected to consume that storage as a static PV, with Persistent Volume Provisioning or as a StorageClass.

For example, many on-premises deployments use Network File System (NFS) to access to some kind of enterprise storage. Persistent volumes are provisioned in Kubernetes like all things in Kubernetes: by means of a manifest. You can learn about creating volumes and persistent volumes in the Kubernetes documentation.

You or your Kubernetes administrators will be responsible for configuring the PVs you create to be consumable as static PV's, with Persistent Volume Provisioning or as a StorageClass.

What you'll need for Pachyderm configuration of PV storage

Keep the information below at hand for when you run `pachctl deploy custom` further on

Configuring with static volumes

You'll need the name of the PV and the amount of space you can use, in gigabytes. We'll refer to those, respectively, as `PVC_STORAGE_NAME` and `PVC_STORAGE_SIZE` further on. With this kind of PV, you'll use the flag `--static-etcd-volume` with `PVC_STORAGE_NAME` as its argument in your deployment.

Note: this will override any attempt to configure with StorageClasses, below.

Configuring with StatefulSets

If you're deploying using *StatefulSets*, you'll just need the amount of space you can use, in gigabytes, which we'll refer to as `PVC_STORAGE_SIZE` further on..

Note: The `--etcd-storage-class` flag and argument will be ignored if you use the flag `--static-etcd-volume` along with it.

Configuring with StatefulSets using StorageClasses

If you're deploying using *StatefulSets* with *StorageClasses*, you'll need the name of the storage class and the amount of space you can use, in gigabytes. We'll refer to those, respectively, as `PVC_STORAGECLASS` and `PVC_STORAGE_SIZE` further on. With this kind of PV, you'll use the flag `--etcd-storage-class` with `PVC_STORAGECLASS` as its argument in your deployment.

Note: The `--etcd-storage-class` flag and argument will be ignored if you use the flag `--static-etcd-volume` along with it.

25.4.3 Deploying an object store

Object store: what's it for?

An object store is used by Pachyderm's `pachd` for storing all your data. The object store you use must be accessible via a low-latency, high-bandwidth connection like Gigabit or 10G Ethernet.

For an on-premises deployment, it's not advisable to use a cloud-based storage mechanism. Don't deploy an on-premises Pachyderm cluster against cloud-based object stores such as S3 from AWS, GCS from Google Cloud Platform, Azure Blob Storage from Azure.

Object store prerequisites

Object stores are accessible using the S3 protocol, created by Amazon. Storage providers like MinIO, EMC's ECS, or SwiftStack provide S3-compatible access to enterprise storage for on-premises deployment. You can find links to instructions for providers of particular object stores in the *See also* section.

Sizing the object store

Size your object store generously. Once you start using Pachyderm, you'll start versioning all your data. We're currently developing good rules of thumb for scaling your object store as your Pachyderm deployment grows, but it's a good idea to start with a large multiple of your current data set size.

What you'll need for Pachyderm configuration of the object store

You'll need four items to configure the object store. We're prefixing each item with how we'll refer to it further on.

1. `OS_ENDPOINT` : The access endpoint. For example, MinIO's endpoints are usually something like `minio-server:9000` . Don't begin it with the protocol; it's an endpoint, not an url.
2. `OS_BUCKET_NAME` : The bucket name you're dedicating to Pachyderm. Pachyderm will need exclusive access to this bucket.
3. `OS_ACCESS_KEY_ID` : The access key id for the object store. This is like a user name for logging into the object store.
4. `OS_SECRET_KEY` : The secret key for the object store. This is like the above user's password.

Keep this information handy.

25.4.4 Next step: creating a custom deploy manifest for Pachyderm

Once you have Kubernetes deployed, your persistent volume created, and your object store configured, it's time to create the Pachyderm manifest for deploying to Kubernetes.

25.5 See Also

25.5.1 Kubernetes variants

- OpenShift

25.5.2 Object storage variants

- EMC ECS
- MinIO
- SwiftStack

CHAPTER 26

Custom Object Stores

In other sections of this guide was have demonstrated how to deploy Pachyderm in a single cloud using that cloud's object store offering. However, Pachyderm can be backed by any object store, and you are not restricted to the object store service provided by the cloud in which you are deploying.

As long as you are running an object store that has an S3 compatible API, you can easily deploy Pachyderm in a way that will allow you to back Pachyderm by that object store. For example, we have seen Pachyderm be backed by Minio, GlusterFS, Ceph, and more.

To deploy Pachyderm with your choice of object store in Google, Azure, or AWS, see the below guides. To deploy Pachyderm on premise with a custom object store, see the on premise docs.

26.1 Common Prerequisites

1. A working Kubernetes cluster and `kubectl`.
2. An account on or running instance of an object store with an S3 compatible API. You should be able to get an ID, secret, bucket name, and endpoint that point to this object store.

26.2 Google + Custom Object Store

Additional prerequisites:

- Google Cloud SDK >= 124.0.0 - If this is the first time you use the SDK, make sure to follow the quick start guide.

First, we need to create a persistent disk for Pachyderm's metadata:

```
# Name this whatever you want, we chose pach-disk as a default
$ STORAGE_NAME=pach-disk

# For a demo you should only need 10 GB. This stores PFS metadata. For reference, 1GB
# should work for 1000 commits on 1000 files.
$ STORAGE_SIZE=[the size of the volume that you are going to create, in GBs. e.g. "10
↪"]

# Create the disk.
gcloud compute disks create --size=${STORAGE_SIZE}GB ${STORAGE_NAME}
```

Then we can deploy Pachyderm:

```
pachctl deploy custom --persistent-disk google --object-store s3 ${STORAGE_NAME} $
↪{STORAGE_SIZE} <object store bucket> <object store id> <object store secret>
↪<object store endpoint> --static-etcd-volume=${STORAGE_NAME}
```

26.3 AWS + Custom Object Store

Additional prerequisites:

- AWS CLI - have it installed and have your AWS credentials configured.

First, we need to create a persistent disk for Pachyderm's metadata:

```
# We recommend between 1 and 10 GB. This stores PFS metadata. For reference 1GB
# should work for 1000 commits on 1000 files.
$ STORAGE_SIZE=[the size of the EBS volume that you are going to create, in GBs. e.g.
↪"10"]

$ AWS_REGION=[the AWS region of your Kubernetes cluster. e.g. "us-west-2" (not us-
↪west-2a)]

$ AWS_AVAILABILITY_ZONE=[the AWS availability zone of your Kubernetes cluster. e.g.
↪"us-west-2a"]

# Create the volume.
$ aws ec2 create-volume --size ${STORAGE_SIZE} --region ${AWS_REGION} --availability-
↪zone ${AWS_AVAILABILITY_ZONE} --volume-type gp2

# Store the volume ID.
$ aws ec2 describe-volumes
$ STORAGE_NAME=[volume id]
```

The we can deploy Pachyderm:

```
pachctl deploy custom --persistent-disk aws --object-store s3 ${STORAGE_NAME} $
↪{STORAGE_SIZE} <object store bucket> <object store id> <object store secret>
↪<object store endpoint> --static-etcd-volume=${STORAGE_NAME}
```

26.4 Azure + Custom Object Store

Additional prerequisites:

- Install Azure CLI >= 2.0.1
- Install jq
- Clone github.com/pachyderm/pachyderm and work from the root of that project.

First, we need to create a persistent disk for Pachyderm's metadata. To do this, start by declaring some environmental variables:

```
# Needs to be globally unique across the entire Azure location
$ RESOURCE_GROUP=[The name of the resource group where the Azure resources will be␣
↪organized]

$ LOCATION=[The Azure region of your Kubernetes cluster. e.g. "West US2"]
```

```
# Needs to be globally unique across the entire Azure location
$ STORAGE_ACCOUNT=[The name of the storage account where your data will be stored]

# Needs to end in a ".vhd" extension
$ STORAGE_NAME=pach-disk.vhd

# We recommend between 1 and 10 GB. This stores PFS metadata. For reference 1GB
# should work for 1000 commits on 1000 files.
$ STORAGE_SIZE=[the size of the data disk volume that you are going to create, in GBs.
↪ e.g. "10"]
```

And then run:

```
# Create a resource group
$ az group create --name=${RESOURCE_GROUP} --location=${LOCATION}

# Create azure storage account
az storage account create \
  --resource-group="${RESOURCE_GROUP}" \
  --location="${LOCATION}" \
  --sku=Standard_LRS \
  --name="${STORAGE_ACCOUNT}" \
  --kind=Storage

# Build microsoft tool for creating Azure VMs from an image
$ STORAGE_KEY="$(az storage account keys list \
                 --account-name="${STORAGE_ACCOUNT}" \
                 --resource-group="${RESOURCE_GROUP}" \
                 --output=json \
                 | jq .[0].value -r
              )"
$ make docker-build-microsoft-vhd
$ VOLUME_URI="$(docker run -it microsoft_vhd \
                "${STORAGE_ACCOUNT}" \
                "${STORAGE_KEY}" \
                "${CONTAINER_NAME}" \
                "${STORAGE_NAME}" \
                "${STORAGE_SIZE}G"
             )"
```

To check that everything has been setup correctly, try:

```
$ az storage account list | jq '.[].name'
```

The we can deploy Pachyderm:

```
pachctl deploy custom --persistent-disk azure --object-store s3 ${VOLUME_URI} $
↪{STORAGE_SIZE} <object store bucket> <object store id> <object store secret>
↪<object store endpoint> --static-etcd-volume=${VOLUME_URI}
```

CHAPTER 27

AWS CloudFront

To deploy a production ready AWS cluster with CloudFront:

1. *Deploy a cloudfront enabled Pachyderm cluster in AWS*
2. *Obtain a cloudfront keypair*
3. *Apply the security credentials*
4. *Verify the setup*

27.1 Deploy a cloudfront enabled cluster in AWS

You'll need to use our "one shot" AWS deployment script to deploy this cluster as follows:

```
$ curl -o aws.sh https://raw.githubusercontent.com/pachyderm/pachyderm/master/etc/
↪deploy/aws.sh
$ chmod +x aws.sh
$ sudo -E ./aws.sh --region=us-east-1 --zone=us-east-1b --use-cloudfront &> deploy.log
```

Here we've redirected the output to a file. Make sure you keep this file around for reference.

Note: You may see a few extra restarts on your pachd pod. Sometimes it takes a bit before your cloudfront distribution comes online

27.2 Obtain a cloudfront keypair

You will most likely need to Ask your IT department for a cloudfront keypair, because only a root AWS account can generate this keypair. You can pass along this link with instructions.

When you get the keypair, you should receive:

- the private/public key (although you only need the private key)
- the keypair ID (which is usually in the filename)

For example,

```
rsa-APKAXXXXXXXXXXXXXXXX.pem
pk-APKAXXXXXXXXXXXXXXXX.pem
```

Here we see that the Key Pair ID is `APKAXXXXXXXXXXXXXXXX`, and the second file is the private key, which should look similar to the following:

```
$ cat pk-APKAXXXXXXXXXXXXXX.pem
-----BEGIN RSA PRIVATE KEY-----
...
```

27.3 Apply the security credentials

You can now run the following script to apply these security credentials to your cloudfront distribution:

```
$ curl -o secure-cloudfront.sh https://raw.githubusercontent.com/pachyderm/pachyderm/
↪master/etc/deploy/cloudfront/secure-cloudfront.sh
$ chmod +x secure-cloudfront.sh
$ ./secure-cloudfront.sh --region us-west-2 --zone us-west-2c --bucket YYYY-pachyderm-
↪store --cloudfront-distribution-id E1BEBVLIDYTLEV  --cloudfront-keypair-id␣
↪APKAXXXXXXXXXXXXXX --cloudfront-private-key-file ~/Downloads/pk-APKAXXXXXXXXXXXXXX.pem
```

where the values for the `--bucket` and `--cloudfront-distribution-id` flags can be obtained from the `deploy.log` file containing your deployment logs.

You will then need to restart the `pachd` pod in kubernetes for the changes to take effect:

```
$ kubectl scale --replicas=0 deployment/pachd && kubectl scale --replicas=1␣
↪deployment/pachd && kubectl get pod
```

27.4 Verify the setup

To verify the setup, we can look at the pachd logs to confirm usage of the cloudfront credentials:

```
$ kubectl get pod
NAME                        READY     STATUS             RESTARTS   AGE
etcd-0                     1/1       Running            0          19h
etcd-1                     1/1       Running            0          19h
etcd-2                     1/1       Running            0          19h
pachd-2796595787-9x0qf     1/1       Running            0          16h
$ kubectl logs pachd-2796595787-9x0qf | grep cloudfront
2017-06-09T22:56:27Z INFO  AWS deployed with cloudfront distribution at d3j9kenawdv8p0
2017-06-09T22:56:27Z INFO  Using cloudfront security credentials - keypair ID␣
↪(APKAXXXXXXXXX) - to sign cloudfront URLs
```

CHAPTER 28

Non-Default Namespaces

Often, production deploys of Pachyderm involve deploying Pachyderm to a non-default namespace. This helps administrators of the cluster more easily manage Pachyderm components alongside other things that might be running inside of Kubernetes (DataDog, TensorFlow Serving, etc.).

To deploy Pachyderm to a non-default namespace, you just need to create that namespace with `kubectl` and then add the `--namespace` flag to your deploy command:

```
$ kubectl create namespace pachyderm
$ pachctl deploy <args> --namespace pachyderm
```

After the Pachyderm pods are up and running, you should see something similar to:

```
$ kubectl get pods --namespace pachyderm
NAME                      READY     STATUS    RESTARTS   AGE
dash-68578d4bb4-mmtbj     2/2       Running   0          3m
etcd-69fcfb5fcf-dgc8j     1/1       Running   0          3m
pachd-784bdf7cd7-7dzxr    1/1       Running   0          3m
```

Note - When using a non-default namespace for Pachyderm, you will have to use the `--namespace` flag for various other `pachctl` command for them to work as expected. These include port-forwarding and undeploy:

```
# forward Pachyderm ports when it was deployed to a non-default namespace
$ pachctl port-forward --namespace pachyderm &

# undeploying Pachyderm when it was deployed to a non-default namespace
$ pachctl undeploy --namespace pachyderm
```

Alternatively or additionally, you might want to set a context similar to the following at the Kubernetes level, such that you can get Pachyderm logs, pod statuses, etc. easily via `kubectl logs ...`, `kubectl get pods`, etc.:

```
$ kubectl config set-context pach --namespace=<pachyderm namespace> \
  --cluster=<cluster> \
  --user=<user>
```

CHAPTER 29

RBAC

Pachyderm has support for Kubernetes Role-Based Access Controls (RBAC) and is a default part of all Pachyderm deployments. For most users, you shouldn't have any issues as Pachyderm takes care of setting all the RBAC permissions automatically. However, if you are deploying Pachyderm on a cluster that your company owns, security policies might not allow certain RBAC permissions by default. Therefore, it's suggested that you contact your Kubernetes admin and provide the following to ensure you don't encounter any permissions issues:

Pachyderm Permission Requirements

```
Rules: []rbacv1.PolicyRule{{
    APIGroups: []string{""},
    Verbs:     []string{"get", "list", "watch"},
    Resources: []string{"nodes", "pods", "pods/log", "endpoints"},
}, {
    APIGroups: []string{""},
    Verbs:     []string{"get", "list", "watch", "create", "update", "delete"},
    Resources: []string{"replicationcontrollers", "services"},
}, {
    APIGroups:     []string{""},
    Verbs:         []string{"get", "list", "watch", "create", "update", "delete"},
    Resources:     []string{"secrets"},
    ResourceNames: []string{client.StorageSecretName},
}},
```

29.1 RBAC and DNS

Kubernetes currently (as of 1.8.0) has a bug that prevents kube-dns from working with RBAC. Not having DNS will make Pachyderm effectively unusable. You can tell if you're being affected by the bug like so:

```
$ kubectl get all --namespace=kube-system
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/kube-dns    1         1         1            0           3m

NAME                        DESIRED   CURRENT   READY     AGE
rs/kube-dns-86f6f55dd5     1         1         0         3m

NAME                              READY     STATUS    RESTARTS   AGE
po/kube-addon-manager-oryx        1/1       Running   0          3m
po/kube-dns-86f6f55dd5-xksnb      2/3       Running   4          3m
po/kubernetes-dashboard-bzjjh     1/1       Running   0          3m
po/storage-provisioner            1/1       Running   0          3m
```

```
NAME                         DESIRED   CURRENT   READY     AGE
rc/kubernetes-dashboard      1         1         1         3m

NAME                          TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)        ␣
↪AGE
svc/kube-dns                  ClusterIP   10.96.0.10     <none>        53/UDP,53/TCP   3m
svc/kubernetes-dashboard      NodePort    10.97.194.16   <none>        80:30000/TCP    3m
```

Notice how `po/kubernetes-dashboard-bzjjh` only has 2/3 pods ready and has 4 restarts. To fix this do:

```
kubectl -n kube-system create sa kube-dns
kubectl -n kube-system patch deploy/kube-dns -p '{"spec": {"template": {"spec": {
↪"serviceAccountName": "kube-dns"}}}}'
```

this will tell Kubernetes that `kube-dns` should use the appropriate ServiceAccount. Kubernetes creates the ServiceAccount, it just doesn't actually use it.

29.2 RBAC Permissions on GKE

If you're deploying Pachyderm on GKE and run into the following error:

```
Error from server (Forbidden): error when creating "STDIN": clusterroles.rbac.
↪authorization.k8s.io "pachyderm" is forbidden: attempt to grant extra privileges:
```

Run the following and redeploy Pachyderm:

```
kubectl create clusterrolebinding cluster-admin-binding --clusterrole=cluster-admin --
↪user=$(gcloud config get-value account)
```

CHAPTER 30

Troubleshooting Deployments

Here are some common issues by symptom related to certain deploys.

- *General Pachyderm cluster deployment*
- Environment-specific
  - *AWS*
    - *Can't connect to the Pachyderm cluster after a rolling update*
    - *The one shot deploy script, `aws.sh`, never completes*
    - *VPC limit exceeded*
    - *GPU node never appears*
  - Google - coming soon...
  - Azure - coming soon...

30.1 General Pachyderm cluster deployment

- *Pod stuck in `CrashLoopBackoff`*
- *Pod stuck in `CrashLoopBackoff` - with error attaching volume*
- [

30.1.1 Pod stuck in `CrashLoopBackoff`

Symptoms

The pachd pod keeps crashing/restarting:

```
$ kubectl get all
NAME                           READY     STATUS             RESTARTS   AGE
po/etcd-281005231-qlkzw        1/1       Running            0          7m
po/pachd-1333950811-0sm1p      0/1       CrashLoopBackOff   6          7m

NAME               CLUSTER-IP       EXTERNAL-IP   PORT(S)                        AGE
svc/etcd           100.70.40.162    <nodes>       2379:30938/TCP                 7m
```

```
svc/kubernetes   100.64.0.1       <none>        443/TCP                        9m
svc/pachd        100.70.227.151   <nodes>       650:30650/TCP,651:30651/TCP    7m

NAME            DESIRED    CURRENT    UP-TO-DATE   AVAILABLE   AGE
deploy/etcd     1          1          1            1           7m
deploy/pachd    1          1          1            0           7m

NAME                   DESIRED    CURRENT    READY      AGE
rs/etcd-281005231      1          1          1          7m
rs/pachd-1333950811    1          1          0          7m
```

Recourse

First describe the pod:

```
$ kubectl describe po/pachd-1333950811-0sm1p
```

If you see an error including `Error attaching EBS volume` or similar, see the recourse for that error here under the corresponding section below. If you don't see that error, but do see something like:

```
  1m     3s     9     {kubelet ip-172-20-48-123.us-west-2.compute.internal}              ␣
↪  Warning     FailedSync     Error syncing pod, skipping: failed to "StartContainer"␣
↪for "pachd" with CrashLoopBackOff: "Back-off 2m40s restarting failed␣
↪container=pachd pod=pachd-1333950811-0sm1p_default(a92b6665-506a-11e7-8e07-
↪02e3d74c49ac)"
```

it means Kubernetes tried running `pachd`, but `pachd` generated an internal error. To see the specifics of this internal error, check the logs for the `pachd` pod:

```
$kubectl logs po/pachd-1333950811-0sm1p
```

Note: If you're using a log aggregator service (e.g. the default in GKE), you won't see any logs when using `kubectl logs ...` in this way. You will need to look at your logs UI (e.g. in GKE's case the stackdriver console).

These logs will most likely reveal the issue directly, or at the very least, a good indicator as to what's causing the problem. For example, you might see, `BucketRegionError: incorrect region,the bucket is not in 'us-west-2' region`. In that case, your object store bucket in a different region than your pachyderm cluster and the fix would be to recreate the bucket in the same region as your pachydermm cluster.

If the error / recourse isn't obvious from the error message, post the error as well as the `pachd` logs in our Slack channel, or open a GitHub Issue and provide the necessary details prompted by the issue template. Please do be sure provide these logs either way as it is extremely helpful in resolving the issue.

30.1.2 Pod stuck in `CrashLoopBackoff` - with error attaching volume

Symptoms

A pod (could be the `pachd` pod or a worker pod) fails to startup, and is stuck in `CrashLoopBackoff`. If you execute `kubectl describe po/pachd-xxxx`, you'll see an error message like the following at the bottom of the output:

```
  30s        30s        1     {attachdetach }                    Warning      ␣
↪FailedMount    Failed to attach volume "etcd-volume" on node "ip-172-20-44-17.us-
↪west-2.compute.internal" with: Error attaching EBS volume "vol-0c1d403ac05096dfe"␣
↪to instance "i-0a12e00c0f3fb047d": VolumeInUse: vol-0c1d403ac05096dfe is already␣
↪attached to an instance
```

```
```

This would indicate that the peristent volume claim is failing to get attached to the node in your kubernetes cluster.

Recourse

Your best bet is to manually detach the volume and restart the pod.

For example, to resolve this issue when Pachyderm is deployed to AWS, pull up your AWS web console and look up the node mentioned in the error message (ip-172-20-44-17.us-west-2.compute.internal in our case). Then on the bottom pane for the attached volume. Follow the link to the attached volume, and detach the volume. You may need to "Force Detach" it.

Once it's detached (and marked as available). Restart the pod by killing it, e.g:

```
$kubectl delete po/pachd-xxx
```

It will take a moment for a new pod to get scheduled.

30.2 AWS Deployment

30.2.1 Can't connect to the Pachyderm cluster after a rolling update

Symptom

After running `kops rolling-update` , `kubectl` (and/or `pachctl`) all requests hang and you can't connect to the cluster.

Recourse

First get your cluster name. You can easily locate that information by running `kops get clusters` . If you used the one shot deployment](http://docs.pachyderm.io/en/latest/deployment/amazon_web_services.html#one-shot-script), you can also get this info in the deploy logs you created by `aws.sh` .

Then you'll need to grab the new public IP address of your master node. The master node will be named something like `master-us-west-2a.masters.somerandomstring.kubernetes.com`

Update the etc hosts entry in `/etc/hosts` such that the api endpoint reflects the new IP, e.g:

```
54.178.87.68 api.somerandomstring.kubernetes.com
```

30.2.2 One shot script never completes

Symptom

The `aws.sh` one shot deploy script hangs on the line:

```
Retrieving ec2 instance list to get k8s master domain name (may take a minute)
```

If it's been more than 10 minutes, there's likely an error.

Recourse

Check the AWS web console / autoscale group / activity history. You have probably hit an instance limit. To confirm, open the AWS web console for EC2 and check to see if you have any instances with names like:

```
master-us-west-2a.masters.tfgpu.kubernetes.com
nodes.tfgpu.kubernetes.com
```

If you don't see instances similar to the ones above the next thing to do is to navigate to "Auto Scaling Groups" in the left hand menu. Then find the ASG with your cluster name:

```
master-us-west-2a.masters.tfgpu.kubernetes.com
```

Look at the "Activity History" in the lower pane. More than likely, you'll see a "Failed" error message describing why it failed to provision the VM. You're probably run into an instance limit for your account for this region. If you're spinning up a GPU node, make sure that your region supports the instance type you're trying to spin up.

A successful provisioning message looks like:

```
Successful
Launching a new EC2 instance: i-03422f3d32658e90c
2017 June 13 10:19:29 UTC-7
2017 June 13 10:20:33 UTC-7
Description:DescriptionLaunching a new EC2 instance: i-03422f3d32658e90c
Cause:CauseAt 2017-06-13T17:19:15Z a user request created an AutoScalingGroup␣
↪changing the desired capacity from 0 to 1. At 2017-06-13T17:19:28Z an instance was␣
↪started in response to a difference between desired and actual capacity, increasing␣
↪the capacity from 0 to 1.
```

While a failed one looks like:

```
Failed
Launching a new EC2 instance
2017 June 12 13:21:49 UTC-7
2017 June 12 13:21:49 UTC-7
Description:DescriptionLaunching a new EC2 instance. Status Reason: You have␣
↪requested more instances (1) than your current instance limit of 0 allows for the␣
↪specified instance type. Please visit http://aws.amazon.com/contact-us/ec2-request␣
↪to request an adjustment to this limit. Launching EC2 instance failed.
Cause:CauseAt 2017-06-12T20:21:47Z an instance was started in response to a␣
↪difference between desired and actual capacity, increasing the capacity from 0 to 1.
```

30.2.3 VPC Limit Exceeded

Symptom

When running `aws.sh` or otherwise deploying with `kops`, you will see:

```
W0426 17:28:10.435315   26463 executor.go:109] error running task "VPC/5120cf0c-
↪pachydermcluster.kubernetes.com" (3s remaining to succeed): error creating VPC:␣
↪VpcLimitExceeded: The  maximum number of VPCs has been reached.
```

Recourse

You'll need to increase your VPC limit or delete some existing VPCs that are not in use. On the AWS web console navigate to the VPC service. Make sure you're in the same region where you're attempting to deploy.

It's not uncommon (depending on how you tear down clusters) for the VPCs not to be deleted. You'll see a list of VPCs here with cluster names, e.g. `aee6b566-pachydermcluster.kubernetes.com`. For clusters that you know are no longer in use, you can delete the VPC here.

30.2.4 GPU Node Never Appears

Symptom

After running `kops edit ig gpunodes` and `kops update` (as outlined here) the GPU node never appears, which can be confirmed via the AWS web console.

Recourse

It's likely you have hit an instance limit for the GPU instance type you're using, or it's possible that AWS doesn't support that instance type in the current region.

Follow these instructions to check for and update Instance Limits. If this region doesn't support your instance type, you'll see an error message like:

```
Failed
Launching a new EC2 instance
2017 June 12 13:21:49 UTC-7
2017 June 12 13:21:49 UTC-7
Description:DescriptionLaunching a new EC2 instance. Status Reason: You have␣
↪requested more instances (1) than your current instance limit of 0 allows for the␣
↪specified instance type. Please visit http://aws.amazon.com/contact-us/ec2-request␣
↪to request an adjustment to this limit. Launching EC2 instance failed.
Cause:CauseAt 2017-06-12T20:21:47Z an instance was started in response to a␣
↪difference between desired and actual capacity, increasing the capacity from 0 to 1.
```

CHAPTER 31

Autoscaling a Pachyderm Cluster

There are 2 levels of autoscaling in Pachyderm:

- Pachyderm can scale down workers when they're not in use.
- Cloud providers can scale workers down/up based on resource utilization (most often CPU).

31.1 Pachyderm Autoscaling of Workers

Refer to the scaleDownThreshold field in the pipeline specification. This allows you to specify a time window after which idle workers are removed. If new inputs come in on the pipeline corresponding to those deleted workers, they get scaled back up.

31.2 Cloud Provider Autoscaling

Out of the box, autoscaling at the cloud provider layer doesn't work well with Pachyderm. However, if configure it properly, cloud provider autoscaling can complement Pachyderm autoscaling of workers.

31.2.1 Default Behavior with Cloud Autoscaling

Normally when you create a pipeline, Pachyderm asks the k8s cluster how many nodes are available. Pachyderm then uses that number as the default value for the pipeline's parallelism. (To read more about parallelism, refer to the distributed processing docs).

If you have cloud provider autoscaling activated, it is possible that your number of nodes will be scaled down to a few or maybe even a single node. A pipeline created on this cluster would have a default parallelism will be set to this low value (e.g., 1 or 2). Then, once the autoscale group notices that more nodes are needed, the parallelism of the pipeline won't increase, and you won't actually make effective use of those new nodes.

31.2.2 Configuration of Pipelines to Complement Cloud Autoscaling

The goal of Cloud autoscaling is to:

- To schedule nodes only as the processing demand necessitates it.

The goals of Pachyderm worker autoscaling are:

- To make sure your job uses a maximum amount of parallelism.

- To ensure that you process the job efficiently.

Thus, to accomplish both of these goals, we recommend:

- Setting a `constant`, high level of parallelism. Specifically, setting the constant parallelism to the number of workers you will need when your pipeline is active.
- Setting the `cpu` and/or `mem` resource requirements in the `resource_requests` field on your pipeline.

To determine the right values for `cpu` / `mem`, first set these values rather high. Then use the monitoring tools that come with your cloud provider (or try out our monitoring deployment) so you can see the actual CPU/mem utilization per pod.

31.2.3 Example Scenario

Let's say you have a certain pipeline with a constant parallelism set to 16. Let's also assume that you've set `cpu` to `1.0` and your instance type has 4 cores.

When a commit of data is made to the input of the pipeline, your cluster might be in a scaled down state (e.g., 2 nodes running). After accounting for the pachyderm services (`pachd` and `etcd`), ~6 cores are available with 2 nodes. K8s then schedules 6 of your workers. That accounts for all 8 of the CPUs across the nodes in your instance group. Your autoscale group then notices that all instances are being heavily utilized, and subsequently scales up to 5 nodes total. Now the rest of your workers get spun up (k8s can now schedule them), and your job proceeds.

This type of setup is best suited for long running jobs, or jobs that take a lot of CPU time. Such jobs give the cloud autoscaling mechanisms time to scale up, while still having data that needs to be processed when the new nodes are up and running.

CHAPTER 32

Data Management Best Practices

This document discusses best practices for minimizing the space needed to store your Pachyderm data, increasing the performance of your data processing as related to data organization, and general good ideas when you are using Pachyderm to version/process your data.

32.1 Shuffling files

Certain pipelines simply shuffle files around (e.g., organizing files into buckets). If you find yourself writing a pipeline that does a lot of copying, such as Time Windowing, it probably falls into this category.

The best way to shuffle files, especially large files, is to create **symlinks** in the output directory that point to files in the input directory.

For instance, to move a file `log.txt` to `logs/log.txt`, you might be tempted to write a `transform` like this:

```
cp /pfs/input/log.txt /pfs/out/logs/log.txt
```

However, it's more efficient to create a symlink:

```
ln -s /pfs/input/log.txt /pfs/out/logs/log.txt
```

Under the hood, Pachyderm is smart enough to recognize that the output file simply symlinks to a file that already exists in Pachyderm, and therefore skips the upload altogether.

Note that if your shuffling pipeline only needs the names of the input files but not their content, you can use `empty_files: true`. That way, your shuffling pipeline can skip both the download and the upload. An example for this type of shuffle pipeline is here

32.2 Garbage collection

When a file/commit/repo is deleted, the data is not immediately removed from the underlying storage system (e.g. S3) for performance and architectural reasons. This is similar to how when you delete a file on your computer, the file is not necessarily wiped from disk immediately.

To actually remove the data, you may need to manually invoke garbage collection. The easiest way to do it is through `pachctl garbage-collect`. Currently `pachctl garbage-collect` can only be started when there are no active jobs running. You also need to ensure that there's no ongoing `put file`. Garbage collection puts the cluster into a readonly mode where no new jobs can be created and no data can be added.

32.3 Setting a root volume size

When planning and configuring your Pachyderm deploy, you need to make sure that each node's root volume is big enough to accommodate your total processing bandwidth. Specifically, you should calculate the bandwidth for your expected running jobs as follows:

```
(storage needed per datum) x (number of datums being processed simultaneously) /␣
↪(number of nodes)
```

Here, the storage needed per datum should be the storage needed for the largest "datum" you expect to process anywhere on your DAG plus the size of the output files that will be written for that datum. If your root volume size is not large enough, pipelines might fail when downloading the input. The pod would get evicted and rescheduled to a different node, where the same thing will happen (assuming that node had a similar volume). This scenario is further discussed here.

Sharing GPU Resources

Often times, teams are running big ML models on instances with GPU resources.

GPU instances are expensive! You want to make sure that you're utilizing the GPUs you're paying for!

33.1 Without configuration

To deploy a pipeline that relies on GPU, you'll already have set the `gpu` resource requirement in the pipeline specification. But Pachyderm workers by default are long lived ... the worker is spun up and waits for new input. That works great for pipelines that are processing a lot of new incoming commits.

For ML workflows, especially during the development cycle, you probably will see lower volume of input commits. Which means that you could have your pipeline workers 'taking' the GPU resource as far as k8s is concerned, but 'idling' as far as you're concerned.

Let's use an example.

Let's say your cluster has a single GPU node with 2 GPUs. Let's say you have a pipeline running that requires 1 GPU. You've trained some models, and found the results were surprising. You suspect your feature extraction code, and are delving into debugging that stage of your pipeline. Meanwhile, the worker you've spun up for your GPU training job is sitting idle, but telling k8s it's using the GPU instance.

Now your coworker is actively trying to develop their GPU model with their pipeline. Their model requires 2 GPUs. But your pipeline is still marked as using 1 GPU, so their pipeline can't run!

33.2 Configuring your pipelines to share GPUs

Whenever you have a limited amount of a resource on your cluster (in this case GPU), you want to make sure you've specified how much of that resource you need via the `resource_requests` as part of your pipeline specification. But, you also need to make sure you set the `standby` field to `true` so that if your pipeline is not getting used, the worker pods get spun down and you free the GPU resource.

CHAPTER 34

Backup and Restore

34.1 Contents

34.2 Introduction

Since release 1.7, Pachyderm provides the commands `pachctl extract` and `pachctl restore` to backup and restore the state of a Pachyderm cluster. (Please see *the note below about releases prior to Pachyderm 1.7*.)

The `pachctl extract` command requires that all pipeline and data loading activity into Pachyderm stop before the extract occurs. This enables Pachyderm to create a consistent, point-in-time backup. In this document, we'll talk about how to create such a backup and restore it to another Pachyderm instance.

Extract and restore commands are currently used to migrate between minor and major releases of Pachyderm, so it's important to understand how to perform them properly. In addition, there are a few design points and operational techniques that data engineers should take into consideration when creating complex pachyderm deployments to minimize disruptions to production pipelines.

In this document, we'll take you through the steps to backup and restore a cluster, migrate an existing cluster to a newer minor or major release, and elaborate on some of those design and operations considerations.

34.3 Backup & restore concepts

Backing up Pachyderm involves the persistent volume (PV) that `etcd` uses for administrative data and the object store bucket that holds Pachyderm's actual data. Restoring involves populating that PV and object store with data to recreate a Pachyderm cluster.

34.4 General backup procedure

34.4.1 1. Pause all pipeline and data loading/unloading operations

Pausing pipelines

From the directed acyclic graphs (DAG) that define your pachyderm cluster, stop each pipeline. You can either run a multiline shell command, shown below, or you must, for each pipeline, manually run the `pachctl stop pipeline` command.

`pachctl stop pipeline <pipeline-name>`

You can confirm each pipeline is paused using the `pachctl list pipeline` command

`pachctl list pipeline`

Alternatively, a useful shell script for running `stop pipeline` on all pipelines is included below. It may be necessary to install the utilities used in the script, like `jq` and `xargs`, on your system.

```
pachctl list pipeline --raw \
  | jq -r '.pipeline.name' \
  | xargs -P3 -n1 -I{} pachctl stop pipeline {}
```

It's also a useful practice, for simple to moderately complex deployments, to keep a terminal window up showing the state of all running kubernetes pods.

`watch -n 5 kubectl get pods`

You may need to install the `watch` and `kubectl` commands on your system, and configure `kubectl` to point at the cluster that Pachyderm is running in.

Pausing data loading operations

Input repositories or **input repos** in pachyderm are repositories created with the `pachctl create repo` command. They're designed to be the repos at the top of a directed acyclic graph of pipelines. Pipelines have their own output repos associated with them, and are not considered input repos. If there are any processes external to pachyderm that put data into input repos using any method (the Pachyderm APIs, `pachctl put file`, etc.), they need to be paused. See *Loading data from other sources into pachyderm* below for design considerations for those processes that will minimize downtime during a restore or migration.

Alternatively, you can use the following commands to stop all data loading into Pachyderm from outside processes.

```
# Once you have stopped all running pachyderm pipelines, such as with this command,
# $ pachctl list pipeline --raw \
#   | jq -r '.pipeline.name' \
#   | xargs -P3 -n1 -I{} pachctl stop pipeline {}

# all pipelines in your cluster should be suspended. To stop all
# data loading processes, we're going to modify the pachd Kubernetes service so that
# it only accepts traffic on port 30649 (instead of the usual 30650). This way,
# any background users and services that send requests to your Pachyderm cluster
# while 'extract' is running will not interfere with the process
#
# Backup the Pachyderm service spec, in case you need to restore it quickly
$ kubectl get svc/pach -o json >pach_service_backup_30650.json

# Modify the service to accept traffic on port 30649
# Note that you'll likely also need to modify your cloud provider's firewall
# rules to allow traffic on this port
$ kubectl get svc/pachd -o json | sed 's/30650/30649/g' | kc apply -f -

# Modify your environment so that *you* can talk to pachd on this new port
$ export PACHD_ADDRESS="${PACHD_ADDRESS/30650/30649}"

# Make sure you can talk to pachd (if not, firewall rules are a common culprit)
$ pc version
COMPONENT           VERSION
pachctl             1.7.11
pachd               1.7.11
```

34.4.2 2. Extract a pachyderm backup

You can use `pachctl extract` alone or in combination with cloning/snapshotting services.

Using `pachctl extract`

Using the `pachctl extract` command, create the backup you need.

`pachctl extract > path/to/your/backup/file`

You can also use the `-u` or `--url` flag to put the backup directly into an object store.

`pachctl extract --url s3://...`

If you are planning on backing up the object store using its own built-in clone operation, be sure to add the `--no-objects` flag to the `pachctl extract` command.

Using your cloud provider's clone and snapshot services

You should follow your cloud provider's recommendation for backing up these resources. Here are some pointers to the relevant documentation.

Snapshotting persistent volumes

For example, here are official guides on creating snapshots of persistent volumes on Google Cloud Platform, Amazon Web Services (AWS) and Microsoft Azure, respectively:

- Creating snapshots of GCE persistent volumes
- Creating snapshots of Elastic Block Store (EBS) volumes
- Creating snapshots of Azure Virtual Hard Disk volumes

For on-premises Kubernetes deployments, check the vendor documentation for your PV implementation on backing up and restoring.

Cloning object stores

Below, we give an example using the Amazon Web Services CLI to clone one bucket to another, taken from the documentation for that command. Similar commands are available for Google Cloud and Azure blob storage.

```
aws s3 sync s3://mybucket s3://mybucket2
```

For on-premises Kubernetes deployments, check the vendor documentation for your on-premises object store for details on backing up and restoring a bucket.

Combining cloning, snapshots and extract/restore

You can use `pachctl extract` command with the `--no-objects` flag to exclude the object store, and use an object store snapshot or clone command to back up the object store. You can run the two commands at the same time. For example, on Amazon Web Services, the following commands can be run simultaneously.

```
aws s3 sync s3://mybucket s3://mybucket2
```

```
pachctl extract --no-objects --url s3://anotherbucket
```

Use case: minimizing downtime during a migration

The above cloning/snapshotting technique is recommended when doing a migration where minimizing downtime is desirable, as it allows the duplicated object store to be the basis of the upgraded, new cluster instead of requiring Pachyderm to extract the data from object store.

34.4.3 3. Restart all pipeline and data loading operations

Once the backup is complete, restart all paused pipelines and data loading operations.

From the directed acyclic graphs (DAG) that define your pachyderm cluster, start each pipeline. You can either run a multiline shell command, shown below, or you must, for each pipeline, manually run the `pachctl start pipeline` command.

```
pachctl start pipeline <pipeline-name>
```

You can confirm each pipeline is started using the `list pipeline` command

```
pachctl list pipeline
```

A useful shell script for running `start pipeline` on all pipelines is included below. It may be necessary to install the utilities used in the script, like `jq` and `xargs` , on your system.

```
pachctl list pipeline --raw \
  | jq -r '.pipeline.name' \
  | xargs -P3 -n1 -I{} pachctl start pipeline {}
```

If you used the port-changing technique, *above*, to stop all data loading into Pachyderm from outside processes, you should change the ports back.

```
# Once you have restarted all running pachyderm pipelines, such as with this command,
# $ pachctl list pipeline --raw \
#   | jq -r '.pipeline.name' \
#   | xargs -P3 -n1 -I{} pachctl start pipeline {}

# all pipelines in your cluster should be restarted. To restart all data loading
# processes, we're going to change the pachd Kubernetes service so that
# it only accepts traffic on port 30650 again (from 30649).
#
# Backup the Pachyderm service spec, in case you need to restore it quickly
$ kubectl get svc/pach -o json >pach_service_backup_30649.json

# Modify the service to accept traffic on port 30650, again
$ kubectl get svc/pachd -o json | sed 's/30649/30650/g' | kc apply -f -

# Modify your environment so that *you* can talk to pachd on the old port
$ export PACHD_ADDRESS="${PACHD_ADDRESS/30649/30650}"

# Make sure you can talk to pachd (if not, firewall rules are a common culprit)
$ pc version
COMPONENT           VERSION
pachctl             1.7.11
pachd               1.7.11
```

34.5 General restore procedure

34.5.1 Restore your backup to a pachyderm cluster, same version

Spin up a Pachyderm cluster and run `pachctl restore` with the backup you created earlier.

`pachctl restore < path/to/your/backup/file`

You can also use the `-u` or `--url` flag to get the backup directly from the object store you placed it in

`pachctl restore --url s3://...`

34.6 Notes and design considerations

34.6.1 Loading data from other sources into Pachyderm

When writing systems that place data into Pachyderm input repos (see *above* for a definition of 'input repo'), it is important to provide ways of 'pausing' output while queueing any data output requests to be output when the systems are 'resumed'. This allows all Pachyderm processing to be stopped while the extract takes place.

In addition, it is desirable for systems that load data into Pachyderm have a mechanism for replaying a queue from any checkpoint in time. This is useful when doing migrations from one release to another, where you would like to minimize downtime of a production Pachyderm system. After an extract, the old system is kept running with the checkpoint established while a duplicate, upgraded pachyderm cluster is migrated with duplicated data. Transactions that occur while the migrated, upgraded cluster is being brought up are not lost, and can be replayed into this new cluster to reestablish state and minimize downtime.

34.6.2 Note about releases prior to Pachyderm 1.7

Pachyderm 1.7 is the first version to support `extract` and `restore`. To bridge the gap to previous Pachyderm versions, we've made a final 1.6 release, 1.6.10, which backports the `extract` and `restore` functionality to the 1.6 series of releases.

Pachyderm 1.6.10 requires no migration from other 1.6.x versions. You can simply `pachctl undeploy` and then `pachctl deploy` after upgrading `pachctl` to version 1.6.10. After 1.6.10 is deployed you should make a backup using `pachctl extract` and then upgrade `pachctl` again, to 1.7.0. Finally you can `pachctl deploy ...` with `pachctl` 1.7.0 to trigger the migration.

CHAPTER 35

Upgrades and Migrations

As new versions of Pachyderm are released, you may need to update your cluster to get access to bug fixes and new features. These updates fall into two categories, which are covered in detail at the links below:

- Upgrades - An upgrade is moving between point releases within the same major release (e.g. 1.7.2 –> 1.7.3). Upgrades are typically a simple process that require little to no downtime.
- Migrations – A migration what you must perform to move between major releases such as 1.8.7 –> 1.9.0.

Important: Performing an *upgrade* when going between *major releases* may lead to corrupted data. *You must perform a migration when going between major releases!*

Whether you're doing an upgrade or migration, it is recommended you backup Pachyderm prior. That will guarantee you can restore your cluster to its previous, good state.

CHAPTER 36

General Troubleshooting

Here are some common issues by symptom along with steps to resolve them.

36.1 Connecting to a Pachyderm Cluster

36.1.1 Cannot connect via `pachctl` - context deadline exceeded

Symptom

You may be using the environment variable `PACHD_ADDRESS` to specify how `pachctl` talks to your Pachyderm cluster, or you may be forwarding the pachyderm port. In any event, you might see something similar to:

```
$ echo $PACHD_ADDRESS
1.2.3.4:30650
$ pachctl version
COMPONENT           VERSION
pachctl             1.4.8
context deadline exceeded
```

Recourse

It's possible that the connection is just taking a while. Occasionally this can happen if your cluster is far away (deployed in a region across the country). Check your internet connection.

It's also possible that you haven't poked a hole in the firewall to access the node on this port. Usually to do that you adjust a security rule (in AWS parlance a security group). For example, on AWS, if you find your node in the web console and click on it, you should see a link to the associated security group. Inspect that group. There should be a way to "add a rule" to the group. You'll want to enable TCP access (ingress) on port 30650. You'll usually be asked which incoming IPs should be whitelisted. You can choose to use your own, or enable it for everyone (0.0.0.0/0).

36.1.2 Certificate Error When Using Kubectl

Symptom

This can happen on any request using `kubectl` (e.g. `kubectl get all`). In particular you'll see:

```
$ kubectl version
Client Version: version.Info{Major:"1", Minor:"6", GitVersion:"v1.6.4", GitCommit:
↪"d6f433224538d4f9ca2f7ae19b252e6fcb66a3ae", GitTreeState:"clean", BuildDate:"2017-
↪05-19T20:41:24Z", GoVersion:"go1.8.1", Compiler:"gc", Platform:"darwin/amd64"}
Unable to connect to the server: x509: certificate signed by unknown authority
```

Recourse

Check if you're on any sort of VPN or other egress proxy that would break SSL. Also, there is a possibility that your credentials have expired. In the case where you're using GKE and gcloud, renew your credentials via:

```
$ kubectl get all
Unable to connect to the server: x509: certificate signed by unknown authority
$ gcloud container clusters get-credentials my-cluster-name-dev
Fetching cluster endpoint and auth data.
kubeconfig entry generated for my-cluster-name-dev.
$ kubectl config current-context
gke_my-org_us-east1-b_my-cluster-name-dev
```

36.1.3 Uploads/Downloads are Slow

Symptom

Any `pachctl put file` or `pachctl get file` commands are slow.

Recourse

If you do not explicitly set the `PACHD_ADDRESS` environment variable, `pachctl` will default to using port forwarding, which throttles traffic to ~1MB/s. If you need to do large downloads/uploads you should consider using the `PACHD_ADDRESS` variable instead to connect directly to your k8s master node. You'll also want to make sure you've allowed ingress access through any firewalls to your k8s cluster.

CHAPTER 37

Troubleshooting Pipelines

37.1 Introduction

Job failures can occur for a variety of reasons, but they generally categorize into 3 failure types:

1. *User-code-related*: An error in the user code running inside the container or the json pipeline config.
2. *Data-related*: A problem with the input data such as incorrect file type or file name.
3. *System- or infrastructure-related*: An error in Pachyderm or Kubernetes such as missing credentials, transient network errors, or resource constraints (for example, out-of-memory–OOM–killed).

In this document, we'll show you the tools for determining what kind of failure it is. For each of the failure modes, we'll describe Pachyderm's and Kubernetes's specific retry and error-reporting behaviors as well as typical user triaging methodologies.

Failed jobs in a pipeline will propagate information to downstream pipelines with empty commits to preserve provenance and make tracing the failed job easier. A failed job is no longer running.

In this document, we'll describe what you'll see, how Pachyderm will respond, and techniques for triaging each of those three categories of failure.

At the bottom of the document, we'll provide specific troubleshooting steps for *specific scenarios*.

- *Pipeline exists but never runs*
- All your pods / jobs get evicted

37.1.1 Determining the kind of failure

First off, you can see the status of Pachyderm's jobs with `pachctl list job`, which will show you the status of all jobs. For a failed job, use `pachctl inspect job <job-id>` to find out more about the failure. The different categories of failures are addressed below.

37.1.2 User Code Failures

When there's an error in user code, the typical error message you'll see is

```
failed to process datum <UUID> with error: <user code error>
```

This means pachyderm successfully got to the point where it was running user code, but that code exited with a non-zero error code. If any datum in a pipeline fails, the entire job will be marked as failed, but datums that did not fail will not need to be reprocessed on future jobs. You can use `pachctl inspect datum <job-id> <datum-id>` or `pachctl logs` with the `--pipeline`, `--job` or `--datum` flags to get more details.

There are some cases where users may want mark a datum as successful even for a non-zero error code by setting the `transform.accept_return_code` field in the pipeline config .

Retries

Pachyderm will automatically retry user code three (3) times before marking the datum as failed. This mitigates datums failing for transient connection reasons.

Triage

`pachctl logs --job=<job_ID>` or `pachctl logs --pipeline=<pipeline_name>` will print out any logs from your user code to help you triage the issue. Kubernetes will rotate logs occasionally so if nothing is being returned, you'll need to make sure that you have a persistent log collection tool running in your cluster. If you set `enable_stats:true` in your pachyderm pipeline, pachyderm will persist the user logs for you.

In cases where user code is failing, changes first need to be made to the code and followed by updating the pachyderm pipeline. This involves building a new docker container with the corrected code, modifying the pachyderm pipeline config to use the new image, and then calling `pachctl update pipeline -f updated_pipeline_config.json`. Depending on the issue/error, user may or may not want to also include the `--reprocess` flag with `update pipeline`.

37.1.3 Data Failures

When there's an error in the data, this will typically manifest in a user code error such as

```
failed to process datum <UUID> with error: <user code error>
```

This means pachyderm successfully got to the point where it was running user code, but that code exited with a non-zero error code, usually due to being unable to find a file or a path, a misformatted file, or incorrect fields/data within a file. If any datum in a pipeline fails, the entire job will be marked as failed. Datums that did not fail will not need to be reprocessed on future jobs.

Retries

Just like with user code failures, Pachyderm will automatically retry running a datum 3 times before marking the datum as failed. This mitigates datums failing for transient connection reasons.

Triage

Data failures can be triaged in a few different way depending on the nature of the failure and design of the pipeline.

In some cases, where malformed datums are expected to happen occasionally, they can be "swallowed" (e.g. marked as successful using `transform.accept_return_codes` or written out to a "failed_datums" directory and handled within user code). This would simply require the necessary updates to the user code and pipeline config as described above. For cases where your code detects bad input data, a "dead letter queue" design pattern may be needed. Many pachyderm developers use a special directory in each output repo for "bad data" and pipelines with globs for detecting bad data direct that data for automated and manual intervention.

Pachyderm's engineering team is working on changes to the Pachyderm Pipeline System in a future release that may make implementation of design patterns like this easier. Take a look at the pipeline design changes for pachyderm 1.9

If a few files as part of the input commit are causing the failure, they can simply be removed from the HEAD commit with `start commit`, `delete file`, `finish commit`. The files can also be corrected in this manner as well. This method is similar to a revert in Git – the "bad" data will still live in the older commits in Pachyderm, but will not be part of the HEAD commit and therefore not processed by the pipeline.

If the entire commit is bad and you just want to remove it forever as if it never happened, `delete commit` will both remove that commit and all downstream commits and jobs that were created as downstream effects of that input data.

37.1.4 System-level Failures

System-level failures are the most varied and often hardest to debug. We'll outline a few common patterns and triage steps. Generally, you'll need to look at deeper logs to find these errors using `pachctl logs --pipeline=<pipeline_name> --raw` and/or `--master` and `kubectl logs pod <pod_name>`.

Here are some of the most common system-level failures:

- Malformed or missing credentials such that a pipeline cannot connect to object storage, registry, or other external service. In the best case, you'll see `permission denied` errors, but in some cases you'll only see "does not exist" errors (this is common reading from object stores)
- Out-of-memory (OOM) killed or other resource constraint issues such as not being able to schedule pods on available cluster resources.
- Network issues trying to connect Pachd, etcd, or other internal or external resources
- Failure to find or pull a docker image from the registry

Retries

For system-level failures, Pachyderm or Kubernetes will generally continually retry the operation with exponential backoff. If a job is stuck in a given state (e.g. starting, merging) or a pod is in `CrashLoopBackoff`, those are common signs of a system-level failure mode.

Triage

Triaging system failures varies as widely as the issues do themselves. Here are options for the common issues mentioned previously.

- Credentials: check your secrets in k8s, make sure they're added correctly to the pipeline config, and double check your roles/perms within the cluster
- OOM: Increase the memory limit/request or node size for your pipeline. If you are very resource constrained, making your datums smaller to require less resources may be necessary.
- Network: Check to make sure etcd and pachd are up and running, that k8s DNS is correctly configured for pods to resolve each other and outside resources, firewalls and other networking configurations allow k8s components to reach each other, and ingress controllers are configured correctly
- Check your container image name in the pipeline config and image_pull_secret.

37.2 Specific scenarios

37.2.1 All your pods / jobs get evicted

Symptom

Running:

```
$ kubectl get all
```

shows a bunch of pods that are marked `Evicted`. If you `kubectl describe ...` one of those evicted pods, you see an error saying that it was evicted due to disk pressure.

Recourse

Your nodes are not configured with a big enough root volume size. You need to make sure that each node's root volume is big enough to store the biggest datum you expect to process anywhere on your DAG plus the size of the output files that will be written for that datum.

Let's say you have a repo with 100 folders. You have a single pipeline with this repo as an input, and the glob pattern is `/*`. That means each folder will be processed as a single datum. If the biggest folder is 50GB and your pipeline's output is about 3 times as big, then your root volume size needs to be bigger than:

```
50 GB (to accommodate the input) + 50 GB x 3 (to accommodate the output) = 200GB
```

In this case we would recommend 250GB to be safe. If your root volume size is less than 50GB (many defaults are 20GB), this pipeline will fail when downloading the input. The pod may get evicted and rescheduled to a different node, where the same thing will happen.

37.2.2 Pipeline exists but never runs

Symptom

You can see the pipeline via:

```
$ pachctl list pipeline
```

But if you look at the job via:

```
$ pachctl list job
```

It's marked as running with `0/0` datums having been processed. If you inspect the job via:

```
$ pachctl inspect job
```

You don't see any worker set. E.g:

```
Worker Status:
WORKER              JOB                 DATUM               STARTED
...
```

If you do `kubectl get pod` you see the worker pod for your pipeline, e.g:

```
po/pipeline-foo-5-v1-273zc
```

But it's state is `Pending` or `CrashLoopBackoff`.

Recourse

First make sure that there is no parent job still running. Do `pachctl list job | grep yourPipelineName` to see if there are pending jobs on this pipeline that were kicked off prior to your job. A parent job is the job that corresponds to the parent output commit of this pipeline. A job will block until all parent jobs complete.

If there are no parent jobs that are still running, then continue debugging:

Describe the pod via:

```
$kubectl describe po/pipeline-foo-5-v1-273zc
```

If the state is `CrashLoopBackoff`, you're looking for a descriptive error message. One such cause for this behavior might be if you specified an image for your pipeline that does not exist.

If the state is `Pending` it's likely the cluster doesn't have enough resources. In this case, you'll see a `could not schedule` type of error message which should describe which resource you're low on. This is more likely to happen if you've set resource requests (cpu/mem/gpu) for your pipelines. In this case, you'll just need to scale up your resources. If you deployed using `kops`, you'll want to do edit the instance group, e.g. `kops edit ig nodes ...` and up the number of nodes. If you didn't use `kops` to deploy, you can use your cloud provider's auto scaling groups to increase the size of your instance group. Either way, it can take up to 10 minutes for the changes to go into effect.

You can read more about autoscaling here

CHAPTER 38

Examples

38.1 OpenCV Edge Detection

This example does edge detection using OpenCV. This is our canonical starter demo. If you haven't used Pachyderm before, start here. We'll get you started running Pachyderm locally in just a few minutes and processing sample log lines.

Open CV

38.2 Word Count (Map/Reduce)

Word count is basically the "hello world" of distributed computation. This example is great for benchmarking in distributed deployments on large swaths of text data.

Word Count

38.3 Periodic Ingress from a Database

This example pipeline executes a query periodically against a MongoDB database outside of Pachyderm. The results of the query are stored in a corresponding output repository. This repository could be used to drive additional pipeline stages periodically based on the results of the query.

Periodic Ingress from MongoDB

38.4 Lazy Shuffle pipeline

This example demonstrates how lazy shuffle pipeline i.e. a pipeline that shuffles, combines files without downloading/uploading can be created. These types of pipelines are useful for intermediate processing step that aggregates or rearranges data from one or many sources. For more information see

Lazy Shuffle pipeline

38.5 Variant Calling and Joint Genotyping with GATK

This example illustrates the use of GATK in Pachyderm for Germline variant calling and joint genotyping. Each stage of this GATK best practice pipeline can be scaled individually and is automatically triggered as data flows into the top of the pipeline. The example follows this tutorial from GATK, which includes more details about the various stages.

GATK - Variant Calling

38.6 Machine Learning

38.6.1 Iris flower classification with R, Python, or Julia

The "hello world" of machine learning implemented in Pachyderm. You can deploy this pipeline using R, Python, or Julia components, where the pipeline includes the training of a SVM, LDA, Decision Tree, or Random Forest model and the subsequent utilization of that model to perform inferences.

R, Python, or Julia - Iris flower classification

38.6.2 Sentiment analysis with Neon

This example implements the machine learning template pipeline discussed in this blog post. It trains and utilizes a neural network (implemented in Python using Nervana Neon) to infer the sentiment of movie reviews based on data from IMDB.

Neon - Sentiment Analysis

38.6.3 pix2pix with TensorFlow

If you haven't seen pix2pix, check out this great demo. In this example, we implement the training and image translation of the pix2pix model in Pachyderm, so you can generate cat images from edge drawings, day time photos from night time photos, etc.

TensorFlow - pix2pix

38.6.4 Recurrent Neural Network with Tensorflow

Based on this Tensorflow example, this pipeline generates a new Game of Thrones script using a model trained on existing Game of Thrones scripts.

Tensorflow - Recurrent Neural Network

38.6.5 Distributed Hyperparameter Tuning

This example demonstrates how you can evaluate a model or function in a distributed manner on multiple sets of parameters. In this particular case, we will evaluate many machine learning models, each configured uses different sets of parameters (aka hyperparameters), and we will output only the best performing model or models.

Hyperparameter Tuning

38.6.6 Spark Example

This example demonstrates integration of Spark with Pachyderm by launching a Spark job on an existing cluster from within a Pachyderm Job. The job uses configuration info that is versioned within Pachyderm, and stores it's reduced result back into a Pachyderm output repo, maintaining full provenance and version history within Pachyderm, while taking advantage of Spark for computation.

Spark Example

Splitting Data for Distributed Processing

As described in the distributed computing with Pachyderm docs, Pachyderm allows you to parallelize computations over data as long as that data can be split up into multiple "datums." However, in many cases, you might have a data set that you want or need to commit into Pachyderm as a single file, rather than a bunch of smaller files (e.g., one per record) that are easily mapped to datums. In these cases, Pachyderm provides an easy way to automatically split your data set for subsequent distributed computing.

Let's say that we have a data set consisting of information about our users. This data is in CSV format in a single file, `user_data.csv`, with one record per line:

```
$ head user_data.csv
1,cyukhtin0@stumbleupon.com,144.155.176.12
2,csisneros1@over-blog.com,26.119.26.5
3,jeye2@instagram.com,13.165.230.106
4,rnollet3@hexun.com,58.52.147.83
5,bposkitt4@irs.gov,51.247.120.167
6,vvenmore5@hubpages.com,161.189.245.212
7,lcoyte6@ask.com,56.13.147.134
8,atuke7@psu.edu,78.178.247.163
9,nmorrell8@howstuffworks.com,28.172.10.170
10,afynn9@google.com.au,166.14.112.65
```

If we just put this into Pachyderm as a single file, we could not subsequently process each of these user records in parallel as separate "datums" (see this guide for more information on datums and distributed computing). Of course, you could manually separate out each of these user records into separate files before you commit them into the `users` repo or via a pipeline stage dedicated to this splitting task. This would work, but Pachyderm actually makes it much easier for you.

The `put file` API includes an option for splitting up the file into separate datums automatically. You can do this with the `pachctl` CLI tool via the `--split` flag on `put file`. For example, to automatically split the `user_data.csv` file up into separate datums for each line, you could execute the following:

```
$ pachctl put file users@master -f user_data.csv --split line --target-file-datums 1
```

The `--split line` argument specifies that Pachyderm should split this file on lines, and the `--target-file-datums 1` arguments specifies that each resulting file should include at most one "datum" (or one line). Note, that Pachyderm will still show the `user_data.csv` entity to you as one entity in the repo:

```
$ pachctl list file users@master
NAME                 TYPE                SIZE
user_data.csv   dir                 5.346 KiB
```

But, this entity is now a directory containing all of the split records:

```
$ pachctl list file users@master:user_data.csv
NAME                             TYPE                SIZE
user_data.csv/0000000000000000   file                43 B
user_data.csv/0000000000000001   file                39 B
user_data.csv/0000000000000002   file                37 B
user_data.csv/0000000000000003   file                34 B
user_data.csv/0000000000000004   file                35 B
user_data.csv/0000000000000005   file                41 B
user_data.csv/0000000000000006   file                32 B
etc...
```

A pipeline that then takes the repo `users` as input with a glob pattern of `/user_data.csv/*` would process each user record (i.e., each line of the CSV) in parallel.

This is, of course, just one example. Right now, Pachyderm supports this type of splitting on lines or on JSON blobs. Here are a few more examples:

```
# Split a json file on json blobs, putting
# each json blob into it's own file.
$ pachctl put file users@master -f user_data.json --split json --target-file-datums 1

# Split a json file on json blobs, putting
# 3 json blobs into each split file.
$ pachctl put file users@master -f user_data.json --split json --target-file-datums 3

# Split a file on lines, putting each 100
# bytes chunk into the split files.
$ pachctl put file users@master -f user_data.txt --split line --target-file-bytes 100
```

39.1 Specifying Header/Footer

Additionally, if your data has a common header or footer, you can specify these manually via `pachctl put-header` or `pachctl put-footer`. This is helpful for CSV data.

To do this, you'll need to specify the header/footer on the *parent directory* of your data. It's a little "magical", but you're essentially embedding the header/footer into the directory and then Pachyderm will apply that header/footer to all the files in that directory. Below we have an example of splitting a CSV with a header, then setting the header explicitly. Notice that once we've set the header, whenever we get a file under that directory, the header is applied. You can still use glob patterns to get all the data under the directory, and in that case the header is still applied.

```
# Raw CSV
$ cat users.csv
id,name,email
4,alice,aaa@place.com
7,bob,bbb@place.com

# Take the raw CSV data minus the header and split it into multiple files:
$ cat users.csv | tail -n +2 | pachctl put file bar@master:users --split line
Reading from stdin.
$ pachctl list file bar@master
NAME  TYPE SIZE
users dir  42B
$ pachctl list file bar@master:users/
NAME                     TYPE SIZE
/users/0000000000000000 file 22B
```

```
/users/0000000000000001 file 20B
# Before we set the header, we just see the raw data when we issue a 'get file'
$ pachctl get file bar@master:users/0000000000000000
4,alice,aaa@place.com

# Now we take the CSV header and apply it to the directory:
$ cat users.csv | head -n 1 | pachctl put-header bar master users
# Now when we read an individual file, we see the header plus the contents
$ pachctl get file bar@master:users/0000000000000000
id,name,email
4,alice,aaa@place.com

# If you issue a 'get file' on the directory, it returns just the header/footer
$ pachctl get file bar@master:users
id,name,email
# We can get the entire CSV file back with:
$ pachctl get file bar@master:users/*
id,name,email
4,alice,aaa@place.com
7,bob,bbb@place.com

# Delete the existing header:
$ echo "" | pachctl put-header repo branch path -f -
# We've now deleted the header
$ pachctl get file bar@master:users/*
4,alice,aaa@place.com
7,bob,bbb@place.com
```

For more info, such as how to delete a header/footer, see `pachctl put-header --help`.

39.2 PG Dump / SQL Support

You can also ingest data from postgres using split file.

1. Generate your PG Dump file

```
$ pg_dump -t users -f users.pgdump
$ cat users.pgdump
--
-- PostgreSQL database dump
--

-- Dumped from database version 9.5.12
-- Dumped by pg_dump version 9.5.12

SET statement_timeout = 0;
SET lock_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '', false);
SET check_function_bodies = false;
SET client_min_messages = warning;
SET row_security = off;

SET default_tablespace = '';
SET default_with_oids = false;

--
-- Name: users; Type: TABLE; Schema: public; Owner: postgres
--

CREATE TABLE public.users (
    id integer NOT NULL,
    name text NOT NULL,
    saying text NOT NULL
);


ALTER TABLE public.users OWNER TO postgres;

--
-- Data for Name: users; Type: TABLE DATA; Schema: public; Owner: postgres
--

COPY public.users (id, name, saying) FROM stdin;
0   wile E Coyote   ...
1   road runner \\.
\.


--
-- PostgreSQL database dump complete
--
```

1. Ingest SQL data using split file

When you use `pachctl put file --split sql ...` your pg dump file is split into three parts - the header, rows, and the footer. The header contains all the SQL statements in the pg dump that setup the schema and tables. The rows are split into individual files (or if you specify the `--target-file-datums` or `--target-file-bytes` multiple rows per file). The footer contains the remaining SQL statements for setting up the tables.

The header and footer are stored on the directory containing the rows. This way, if you request a `get file` on the directory, you'll get just the header and footer. If you request an individual file, you'll see the header plus the row(s) plus the footer. If you request all the files with a glob pattern, e.g. `/directoryname/*`, you'll receive the header plus all the rows plus the footer, recreating the full pg dump. In this way, you can construct full or partial pg dump files so that you can load full or partial data sets.

```
$ pachctl put file data@master -f users.pgdump --split sql
$ pachctl put file data@master:users --split sql -f users.pgdump
$ pachctl list file data@master
NAME         TYPE SIZE
users        dir  914B
$ pachctl list file data@master:users
NAME                           TYPE SIZE
/users/0000000000000000        file 20B
/users/0000000000000001        file 18B
```

Then in your pipeline (where you've started and forked postgres), you can load the data by doing something like:

```
$ cat /pfs/data/users/* | sudo -u postgres psql
```

And with a glob pattern `/*` this code would load each raw postgres chunk into your postgres instance for processing by your pipeline.

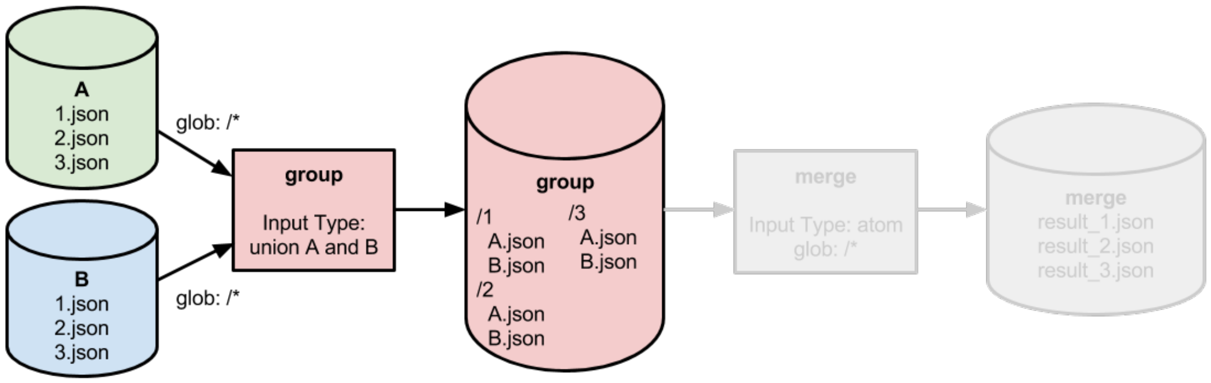
For this use case, you'll likely want to use `--target-file-datums` or `--target-file-bytes` since it's likely that you'll want to run your queries against many rows at a time.

Combining/Merging/Joining Data

There are a variety of use cases in which you would want to match datums from multiple data repositories to do some combined processing, joining, or aggregation. For example, you may need to process multiple records corresponding to a certain user, a certain experiment, or a certain device together. In these scenarios, we recommend a 2-stage method of merging your data:

1. *A first pipeline* that groups all of the records for a specific key/index.
2. *A second pipeline* that takes that grouped output and performs the merging, joining, or other processing for the group.

40.1 1. Grouping records that need to be processed together



Let's say that we have two repositories containing JSON records, `A` and `B`. These repositories may correspond to two experiments, two geographic regions, two different devices generating data, etc. In any event, the repositories look similar to:

```
$ pachctl list file A@master
NAME                TYPE                SIZE
1.json              file                39 B
2.json              file                39 B
3.json              file                39 B
$ pachctl list file B@master
NAME                TYPE                SIZE
1.json              file                39 B
2.json              file                39 B
3.json              file                39 B
```

We need to process `A/1.json` with `B/1.json` to merge their contents or otherwise process them together. Thus, we need to group each set of JSON records into respective "datums" that can each be processed together by our *second pipeline* (read more about datums and distributed processing here).

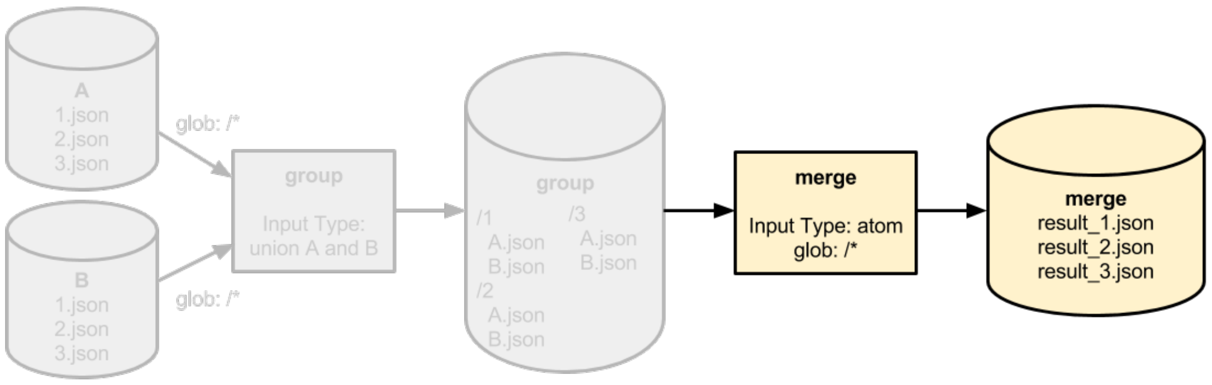
The first pipeline takes a union of `A` and `B` as inputs, each with glob pattern `/*`. As each JSON file is processed, it is copied to a folder in the output corresponding to the key/index for that record (in this case, just the number in the file name). It is also re-named to a unique name corresponding to it's source:

```
/1
  A.json
  B.json
/2
  A.json
  B.json
/3
  A.json
  B.json
```

Note, that when performing this grouping:

- You should use `"lazy": true` to avoid unnecessary downloads of data.
- You should use sym-links to avoid unnecessary uploads of data and unnecessary duplication of data (see more information on "copy elision" here).

40.2 2. Processing the grouped records



Once the records that need to be processed together are grouped by the first pipeline, our second pipeline can take the `group` repository as input with a glob pattern of `/*`. This will let the second pipeline process each grouping of records in parallel.

The second pipeline will perform any merging, aggregation, or other processing on the respective grouping of records and could, for example, output each respective result to the root of the output directory:

```
$ pachctl list file merge@master
NAME                 TYPE                SIZE
result_1.json              file                39 B
result_2.json              file                39 B
result_3.json              file                39 B
```

40.3 Implications and Notes

- This 2-stage pattern of combining data could be used for merging or grouped processing of data from various experiments, devices, etc. However, the same pattern can be applied to perform distributed joins of tabular data or data from database tables. For example, you could join user email records together with user IP records on the key/index of a user ID.
- Each of the 2 stages can be parallelized across workers to scaled with the size of your data and the number of data sources that you are merging.
- In some cases, your data may not be split into separate files for each record. In these cases, you can utilize Pachyderm splitting functionality to prepare your data for this sort of distributed merging/joining.

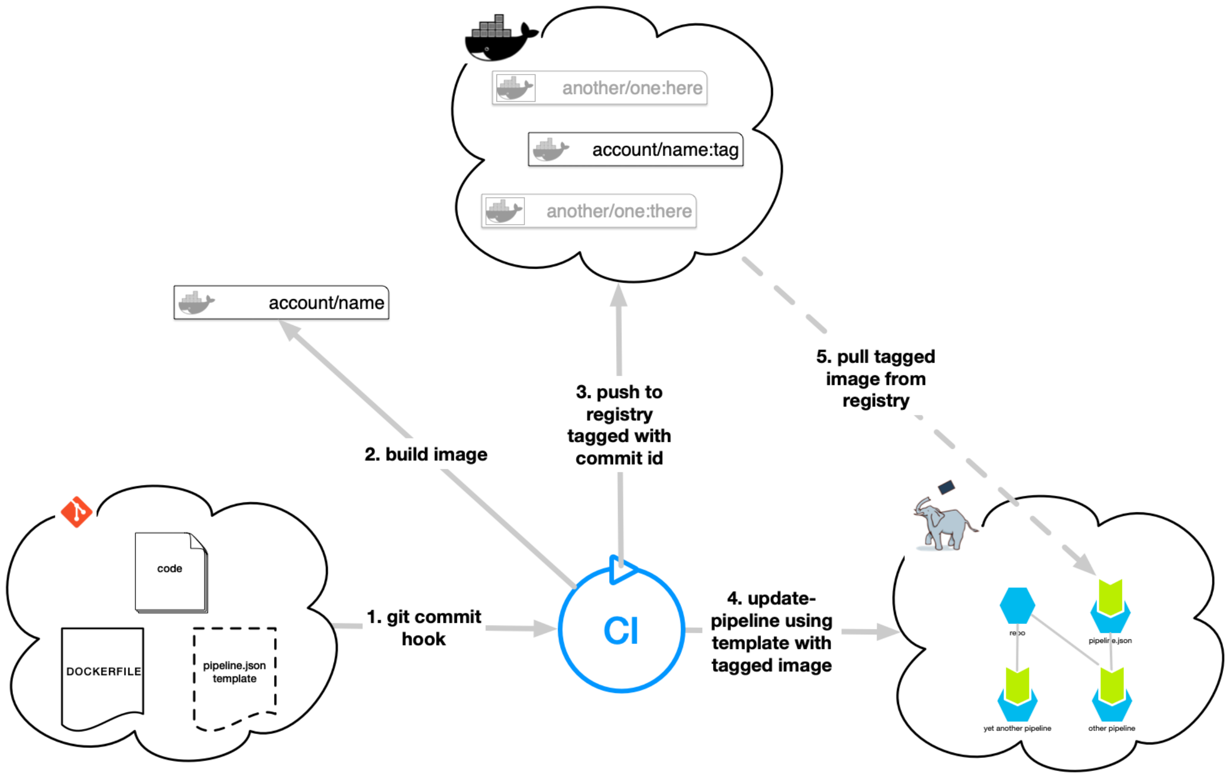
CHAPTER 41

Example Developer Workflow

41.1 Introduction

Pachyderm is a powerful system for providing data provenance and scalable processing to data scientists and engineers. You can make it even more powerful by integrating it with your existing continuous integration and continuous deployment workflows. If your organization is fielding a new, production data science application, this workflow can help you by making it the foundation of new CI/CD processes you establish within your data science and engineering groups. In this document, we'll discuss this basic workflow you can use as the basis for your own workflows.

41.2 Basic workflow



As you write code, you test it in containers and notebooks against sample data in Pachyderm repos. You can also run your code in development pipelines in Pachyderm. Pachyderm provides facilities to help with day-to-day development practices, including the `--build` and `--push-images` flags to the `update pipeline` command, which can build & push or just push images to a docker registry.

There are a couple of things to note about the files shown in git, in the left-hand side of the diagram above. The pipeline.json template file, in addition to being used for CI/CD as noted below, could be used with local build targets in a makefile for development purposes: the local build uses DOCKERFILE and creates a pipeline.json for use in development pipelines. This is optional, of course, but may fit in with some workflows.

Once your code is ready to commit to your git repo, here are the steps that can form the basis of a production workflow.

41.2.1 1. Git commit hook

A commit hook in git for your kicks off the Continous Integration/Continuous Deployment process. It should use the information present in a template for your Pachyderm pipelines for subsequent steps.

41.2.2 2. Build image

Your CI process should automatically kick off the build of an docker container image based on your code and the DOCKERFILE. That image will be used in the next step.

41.2.3 3. Push to registry tagged with commit id

The docker image created in the prior step is then pushed to your preferred docker registry and tagged with the git commit SHA, shown as just "tag" in the figure above.

41.2.4 4. 'update pipeline' using template with tagged image

In this step, your CI/CD infrastructure would use the pipeline.json template that was checked in, and fill in the git commit SHA for the version of the image that should be used in this pipeline. It will then use the `pachctl update pipeline` command to push it to pachyderm.

41.2.5 5. Pull tagged image from registry

Pachyderm handles this part automatically for you, but we include it here for completeness. When the production pipeline is updated with the `pipeline.json` file that has the correct image tag in it, it will automatically restart all pods for this pipeline with the new image.

41.3 Tracking provenance

When looking at a job using the `pachctl inspect job` command, you can see the exact image tag that produced the commits in that job, bridging from data provenance to code provenance.

`pachctl list job` gives you `--input` and `--output` flags that can be used with an argument in the form of repo@branch-or-commit to get you complete provenance on the jobs that produced a particular commit in a particular repo.

41.4 Summary

Pachyderm can provide data provenance and reproducibility to your production data science applications by integrating it with your existing continuous integration and continuous deployment workflows, or creating new workflows using standard technologies.

CHAPTER 42

Triggering Pipelines Periodically (cron)

Pachyderm pipelines are triggered by changes to their input data repositories (as further discussed in What Happens When You Create a Pipeline). However, if a pipeline consumes data from sources outside of Pachyderm, it can't use Pachyderm's triggering mechanism to process updates from those sources. For example, you might need to:

- Scrape websites
- Make API calls
- Query a database
- Retrieve a file from S3 or FTP

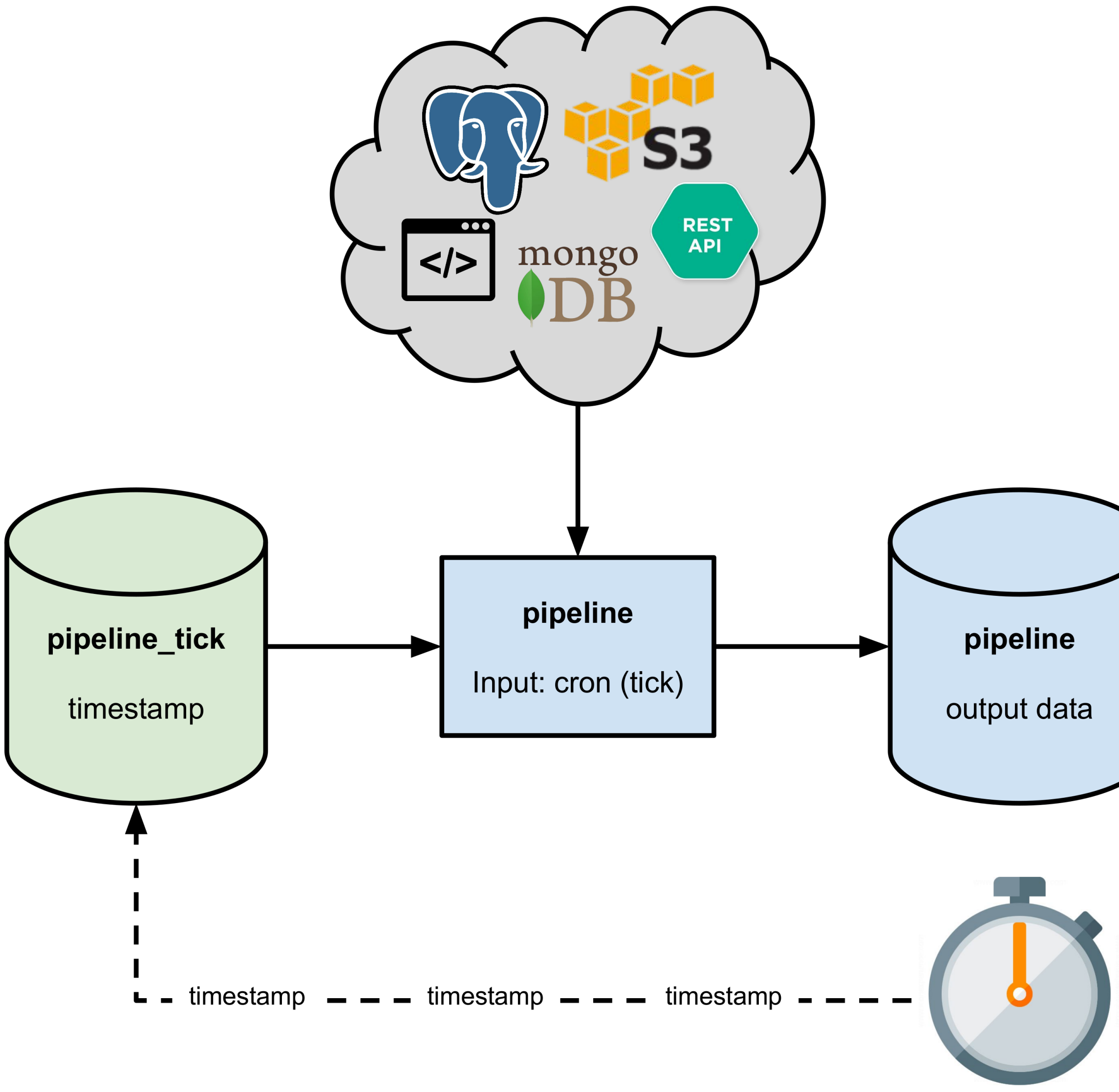
You can schedule pipelines like these to run regularly with Pachyderm's built-in `cron` input type. You can find an example pipeline that queries MongoDB periodically here.

42.1 Cron Example

Let's say that we want to query a database every 10 seconds and update our dataset with the new data every time the pipeline is triggered. We could do this with `cron` input as follows:

```
"input": {
  "cron": {
    "name": "tick",
    "spec": "@every 10s"
  }
}
```

When we create this pipeline, Pachyderm will create a new input data repository corresponding to the `cron` input. It will then automatically commit a timestamp file every 10 seconds to the `cron` input repository, which will automatically trigger our pipeline.



The pipeline will run every 10 seconds, querying our database and updating its output.

We have used the `@every 10s` cron spec here, but you can use any cron spec formatted according to RFC 3339. For example, `*/10 * * * *` would indicate that the pipeline should run every 10 minutes (these time formats should be familiar to those who have used cron in the past, and you can find more examples here)

By default, Pachyderm will run the pipeline on input data that has come in since the last tick. If instead we would like the pipeline to reprocess all the data, we can set the `overwrite` flag to true:

```
"input": {
  "cron": {
    "name": "tick",
    "spec": "@every 10s",
    "overwrite": true
  }
}
```

Now, it will overwrite the timestamp file each tick. Since the processed data is associated with the old file, its absence will indicate to Pachyderm that it needs to be reprocessed.

CHAPTER 43

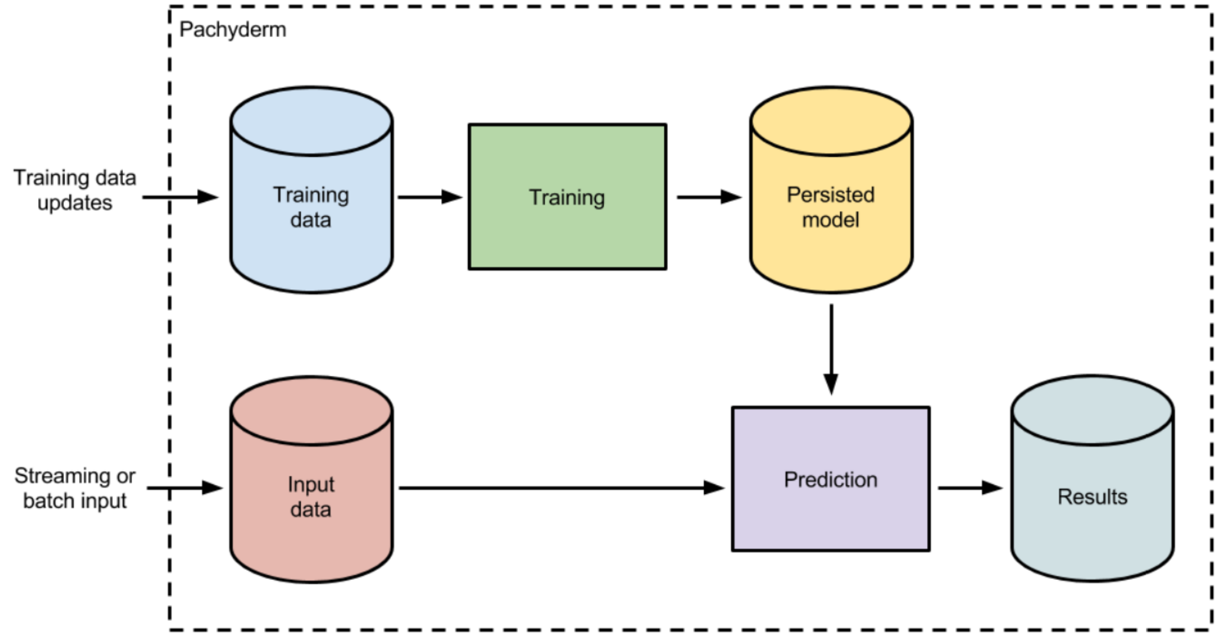
Creating Machine Learning Workflows

Because Pachyderm is language/framework agnostic and because it easily distributes analyses over large data sets, data scientists can use whatever tooling they like for ML. Even if that tooling isn't familiar to the rest of an engineering organization, data scientists can autonomously develop and deploy scalable solutions via containers. Moreover, Pachyderm's pipelining logic paired with data versioning, allows any results to be exactly reproduced (e.g., for debugging or during the development of improvements to a model).

We recommend combining model training processes, persisted models, and a model utilization processes (e.g., making inferences or generating results) into a single Pachyderm pipeline DAG (Directed Acyclic Graph). Such a pipeline allows us to:

- Keep a rigorous historical record of exactly what models were used on what data to produce which results.
- Automatically update online ML models when training data or parameterization changes.
- Easily revert to other versions of an ML model when a new model is not performing or when "bad data" is introduced into a training data set.

This sort of sustainable ML pipeline looks like this:



A data scientist can update the training dataset at any time to automatically train a new persisted model. This training could utilize any language or framework (Spark, Tensorflow, scikit-learn, etc.) and output any format of persisted model (pickle, XML, POJO, etc.). Regardless of framework, the model will be versioned by Pachyderm, and you will be able to track what "Input data" was input into which model AND exactly what "Training data" was used to train that model.

Any new input data coming into the "Input data" repository will be processed with the updated model. Old predictions can be re-computed with the updated model, or new models could be backtested on previously input and versioned data. This will allow you to avoid manual updates to historical results or having to worry about how to swap out ML models in production!

43.1 Examples

We have implemented this machine learning workflow in some example pipelines using a couple of different frameworks. These examples are a great starting point if you are trying to implement ML in Pachyderm.

Processing Time-Windowed Data

If you are analyzing data that is changing over time, chances are that you will want to perform some sort of analysis on "the last two weeks of data," "January's data," or some other moving or static time window of data. There are a few different ways of doing these types of analyses in Pachyderm, depending on your use case. We recommend one of the following patterns for:

1. *Fixed time windows* - for rigid, fixed time windows, such as months (Jan, Feb, etc.) or days (01-01-17, 01-02-17, etc.).
2. *Moving or rolling time windows* - for rolling time windows of data, such as three day windows or two week windows.

44.1 Fixed time windows

As further discussed in Creating Analysis Pipelines and Distributed Computing, the basic unit of data partitioning in Pachyderm is a "datum" which is defined by a glob pattern. When analyzing data within fixed time windows (e.g., corresponding to fixed calendar times/dates), we recommend organizing your data repositories such that each of the time windows that you are going to analyze corresponds to a separate files or directories in your repository. By doing this, you will be able to:

- Analyze each time window in parallel.
- Only re-process data within a time window when that data, or a corresponding data pipeline, changes.

For example, if you have monthly time windows of JSON sales data that need to be analyzed, you could create a `sales` data repository and structure it like:

```
sales
-- January
|   -- 01-01-17.json
|   -- 01-02-17.json
|   -- etc...
-- February
|   -- 01-01-17.json
|   -- 01-02-17.json
|   -- etc...
-- March
    -- 01-01-17.json
    -- 01-02-17.json
    -- etc...
```

When you run a pipeline with an input repo of `sales` having a glob pattern of `/*`, each month's worth of sales data is processed in parallel (if possible). Further, when you add new data into a subset of the months or add data into a new month (e.g., May), only those updated datums will be re-processed.

More generally, this structure allows you to create:

- Pipelines that aggregate, or otherwise process, daily data on a monthly basis via a `/*` glob pattern.
- Pipelines that only analyze a certain month's data via, e.g., a `/January/*` or `/January/` glob pattern.
- Pipelines that process data on a daily basis via a `/*/*` glob pattern.
- Any combination of the above.

44.2 Moving or rolling time windows

In certain use cases, you need to run analyses for moving or rolling time windows, even when those don't correspond to certain calendar months, days, etc. For example, you may need to analyze the last three days of data, the three days of data prior to that, the three days of data prior to that, etc. In other words, you need to run an analysis for every rolling length of time.

For rolling or moving time windows, there are a couple of recommended patterns:

1. Bin your data in repository folders for each of the rolling/moving time windows.
2. Maintain a time windowed set of data corresponding to the latest of the rolling/moving time windows.

44.2.1 Binning data into rolling/moving time windows

In this method of processing rolling time windows, we'll use a two-pipeline DAG to analyze time windows efficiently:

- *Pipeline 1* - Read in data, determine which bins the data corresponds to, and write the data into those bins
- *Pipeline 2* - Read in and analyze the binned data.

By splitting this analysis into two pipelines we can benefit from parallelism at the file level. In other words, *Pipeline 1* can be easily parallelized for each file, and *Pipeline 2* can be parallelized per bin. Now we can scale the pipelines easily as the number of files increases.

Let's take the three day rolling time windows as an example, and let's say that we want to analyze three day rolling windows of sales data. In a first repo, called `sales`, a first day's worth of sales data is committed:

```
sales
-- 01-01-17.json
```

We then create a first pipeline to bin this into a repository directory corresponding to our first rolling time window from 01-01-17 to 01-03-17:

```
binned_sales
-- 01-01-17_to_01-03-17
    -- 01-01-17.json
```

When our next day's worth of sales is committed,

```
sales
-- 01-01-17.json
-- 01-02-17.json
```

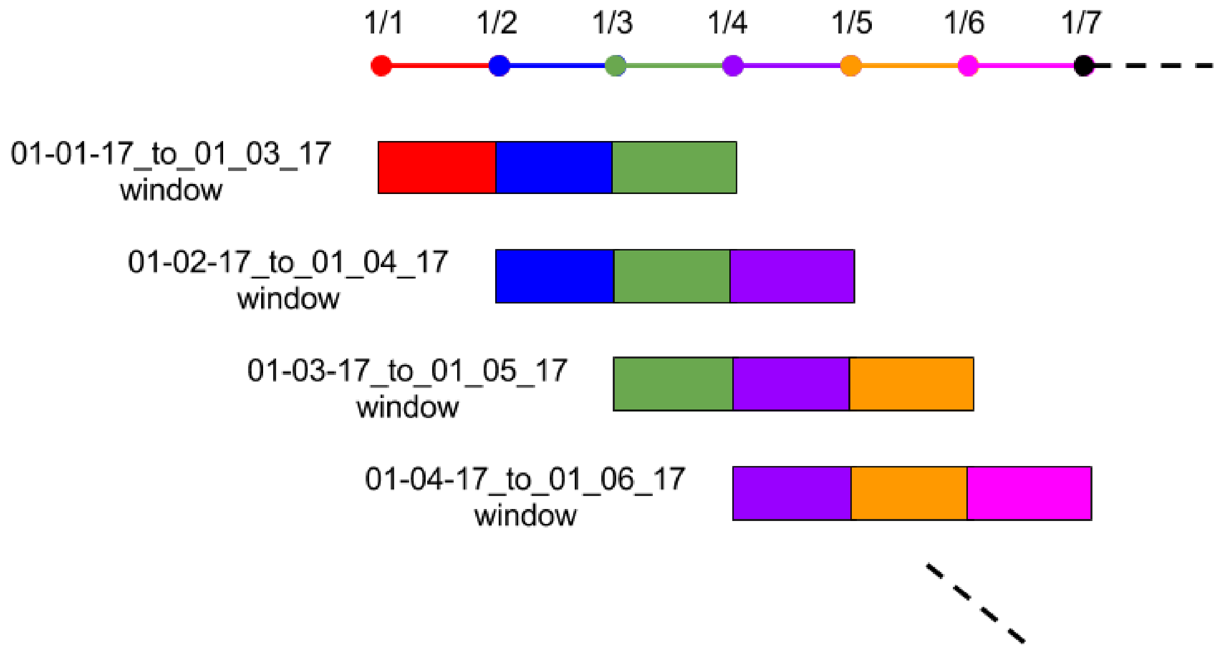
the first pipeline executes again to bin the 01-02-17 data into any relevant bins. In this case, we would put it in the previously created bin for 01-01-17 to 01-03-17, but we would also put it into a bin starting on 01-02-17:

```
binned_sales
-- 01-01-17_to_01-03-17
|   -- 01-01-17.json
|   -- 01-02-17.json
-- 01-02-17_to_01-04-17
    -- 01-02-17.json
```

As more and more daily data is added, you will end up with a directory structure that looks like:

```
binned_sales
-- 01-01-17_to_01-03-17
|   -- 01-01-17.json
|   -- 01-02-17.json
|   -- 01-03-17.json
-- 01-02-17_to_01-04-17
|   -- 01-02-17.json
|   -- 01-03-17.json
|   -- 01-04-17.json
-- 01-03-17_to_01-05-17
|   -- 01-03-17.json
|   -- 01-04-17.json
|   -- 01-05-17.json
-- etc...
```

and is maintained over time as new data is committed:


1/1
1/2
1/3
1/4
1/5
1/6
1/7
01-01-17_to_01_03_17 window
01-02-17_to_01_04_17 window
01-03-17_to_01_05_17 window
01-04-17_to_01_06_17 window

Your second pipeline can then process these bins in parallel, via a glob pattern of `/*`, or in any other relevant way as discussed further in the *"Fixed time windows" section*. Both your first and second pipelines can be easily parallelized.

Note - When looking at the above directory structure, it may seem like there is an unnecessary duplication of the data. However, under the hood Pachyderm deduplicates all of these files and maintains a space efficient representation of your data. The binning of the data is merely a structural re-arrangement to allow you to process these types of rolling time windows.

Note - It might also seem as if there is unnecessary data transfers over the network to perform the above binning. Pachyderm can ensure that performing these types of "shuffles" doesn't actually require transferring data over the network. Read more about that here.

44.2.2 Maintaining a single time-windowed data set

The advantage of the binning pattern above is that any of the rolling time windows are available for processing. They can be compared, aggregated, combined, etc. in any way, and any results or aggregations are kept in sync with updates to the bins. However, you do need to put in some logic to maintain the binning directory structure.

There is another pattern for moving time windows that avoids the binning of the above approach and maintains an up-to-date version of a moving time-windowed data set. It also involves two pipelines:

- *Pipeline 1* - Read in data, determine which files belong in your moving time window, and write the relevant files into an updated version of the moving time-windowed data set.
- *Pipeline 2* - Read in and analyze the moving time-windowed data set.

Let's utilize our sales example again to see how this would work. In the example, we want to keep a moving time window of the last three days worth of data. Now say that our daily `sales` repo looks like the following:

```
sales
-- 01-01-17.json
-- 01-02-17.json
-- 01-03-17.json
-- 01-04-17.json
```

When the January 4th file, `01-04-17.json`, is committed, our first pipeline pulls out the last three days of data and arranges it like so:

```
last_three_days
-- 01-02-17.json
-- 01-03-17.json
-- 01-04-17.json
```

Think of this as a "shuffle" step. Then, when the January 5th file, `01-05-17.json`, is committed,

```
sales
-- 01-01-17.json
-- 01-02-17.json
-- 01-03-17.json
-- 01-04-17.json
-- 01-05-17.json
```

the first pipeline would again update the moving window:

```
last_three_days
-- 01-03-17.json
-- 01-04-17.json
-- 01-05-17.json
```

Whatever analysis we need to run on the moving windowed data set in `moving_sales_window` can use a glob pattern of `/` or `/*` (depending on whether we need to process all of the time windowed files together or they can be processed in parallel).

Warning - When creating this type of moving time-windowed data set, the concept of "now" or "today" is relative. It is important that you make a sound choice for how to define time based on your use case (e.g., by defaulting to UTC). You should not use a function such as `time.now()` to figure out a current day. The actual time at which this analysis is run may vary. If you have further questions about this issue, please do not hesitate to reach out to us via Slack or at support@pachyderm.io.

Ingressing From a Separate Object Store

Occasionally, you might find yourself needing to ingress data from or egress data (with the `put file` command or `egress` field in the pipeline spec) to/from an object store that runs in a different cloud. For instance, you might be running a Pachyderm cluster in Azure, but you need to ingress files from a S3 bucket.

Fortunately, Pachyderm can be configured to ingress/egress from any number of supported cloud object stores, which currently include S3, Azure, and GCS. In general, all you need to do is to provide Pachyderm with the credentials it needs to communicate with the cloud provider.

To provide Pachyderm with the credentials, you use the `pachctl deploy storage` command:

```
$ pachctl deploy storage <backend> ...
```

Here, `<backend>` can be one of `aws` , `google` , and `azure` , and the different backends take different parameters. Execute `pachctl deploy storage <backend>` to view detailed usage information.

For example, here's how you would deploy credentials for a S3 bucket:

```
$ pachctl deploy storage aws <region> <bucket-name> <access key id> <secret access␣
↪key>
```

Credentials are stored in a Kubernetes secret and therefore share the same security properties.

CHAPTER 46

Utilizing GPUs

Pachyderm currently supports GPUs through Kubernetes device plugins. If you already have a GPU enabled Kubernetes cluster through device plugins, then skip to Using GPUs in Pipelines.

46.1 Setting up a GPU Enabled Kubernetes Cluster

For guidance on how to set up a GPU enabled Kubernetes cluster through device plugins, refer to the Kubernetes docs.

Setting up a GPU enabled Kubernetes cluster can be a difficult process depending on the application/framework and hardware being used. Some general things to check for if you are running into issues are:

1. The correct software is installed on the GPU machines such that applications running in Docker containers can use the GPUs. This is going to be highly dependent on the manufacturer of the GPUs and how you are using them. The most straightforward approach is to get a VM image with this pre-installed and/or use management software such as kops (nvidia-device-plugin).
2. Kubernetes is exposing the GPU resources. This can be checked by describing the GPU nodes with `kubectl describe node`. You should see the GPU resources marked as allocatable/scheduleable if they are setup properly.
3. Your application/framework can access and use the GPUs. This may be as simple as making shared libraries accesible by the application/framework running in your container. Which can be done by baking environment variables into the Docker image or passing in environment variables through the pipeline spec.

46.2 Using GPUs in Pipelines

If you already have a GPU enabled Kubernetes cluster through device plugins, then using GPUs in your pipelines is as simple as setting up a GPU resource limit with the type and number of GPUs. An example pipeline spec for a GPU enabled pipeline is as follows:

```
{
  "pipeline": {
    "name": "train"
  },
  "transform": {
    "image": "acme/your-gpu-image",
    "cmd": [
      "python",
      "train.py"
    ],
  },
  "resource_limits": {
    "memory": "1024M",
    "gpu": {
      "type": "nvidia.com/gpu",
      "number": 1
    }
  },
  "inputs": {
    "pfs": {
      "repo": "data",
      "glob": "/*"
    }
  ]
}
```

CHAPTER 47

Deferred Processing of Data

While they're running, Pachyderm Pipelines will process any new data you commit to their input branches. This can be annoying in cases where you want to commit data more frequently than you want to process.

This is generally not an issue because Pachyderm pipelines are smart about not reprocessing things they've already processed, but some pipelines need to process everything from scratch. For example, you may want to commit data every hour, but only want to retrain a machine learning model on that data daily since it needs to train on all the data from scratch.

In these cases there's a massive performance benefit to deferred processing. This document covers how to achieve that and control exactly what gets processed when using the Pachyderm system.

The key thing to understand about controlling when data is processed in Pachyderm is that you control this using the *filesystem*, rather than at the pipeline level. Pipelines are inflexible but simple, they always try to process the data at the heads of their input branches. The filesystem, on the other hand, is much more flexible and gives you the ability to commit data in different places and then efficiently move and rename the data so that it gets processed when you want. The examples below describe how specifically this should work for common cases.

47.1 Using a staging branch

The simplest and most common pattern for deferred processing is using a `staging` branch in addition to the usual `master` branch that the pipeline takes as input. To begin, create your input repo and your pipeline (which by default will read from the master branch). This will automatically create a branch on your input repo called `master`. You can check this with `list branch`:

```
$ pachctl list branch data
BRANCH HEAD
master -
```

Notice that the head commit is empty. This is why the pipeline has no jobs as pipelines process the HEAD commit of their input branches. No HEAD commit means no processing. If you were to commit data to the `master` branch, the pipeline would immediately kick off a job to process what you committed. However, if you want to commit something without immediately processing it you need to commit it to a different branch. That's where a `staging` branch comes in – you're essentially adding your data into a staging area to then process later.

Commit a file to the staging branch:

```
$ pachctl put file data@staging -f <file>
```

Your repo now has 2 branches, `staging` and `master` (`put file` automatically creates branches if they don't exist). If you do `list branch` again you should see:

```
$ pachctl list branch data
BRANCH  HEAD
staging f3506f0fab6e483e8338754081109e69
master  -
```

Notice that `master` still doesn't have a head commit, but the new branch, `staging`, does. There still have been no jobs, because there are no pipelines taking `staging` as inputs. You can continue to commit to `staging` to add new data to the branch and it still won't process anything. True to its name, it's acting as a staging ground for data.

When you're ready to actually process the data all you need to do is update the master branch to point to the head of the staging branch:

```
$ pachctl create branch data@master --head staging
$ pachctl list branch
staging f3506f0fab6e483e8338754081109e69
master  f3506f0fab6e483e8338754081109e69
```

Notice that `master` and `staging` now have the same head commit. This means that your pipeline finally has something to process. If you do `list job` you should see a new job. Notice that even if you created multiple commits on `staging` before updating `master` you still only get 1 job. Despite the fact that those other commits are ancestors of the current HEAD of master, they were never the actual HEAD of `master` themselves, so they don't get processed. This is often fine because commits in Pachyderm are generally additive, so processing the HEAD commit also processes data from previous commits.

However, sometimes you want to process specific intermediary commits. To do this, all you need to do is set `master` to have them as HEAD. For example if you had 10 commits on `staging` and you wanted to process the 7th, 3rd, and most recent commits, you would do:

```
$ pachctl create branch data@master --head staging^7
$ pachctl create branch data@master --head staging^3
$ pachctl create branch data@master --head staging
```

If you do `list job` while running the above commands, you will see between 1 and 3 new jobs. Eventually there will be a job for each of the HEAD commits, however Pachyderm won't create a new job until the previous job has completed.

47.1.1 What to do if you accidentally process something you didn't want to

In all of the examples above we've been *advancing* the `master` branch to later commits. However, this isn't a requirement of the system, you can move backward to previous commits just as easily. For example, if after the above commands you realized that actually want your final output to be the result of processing `staging^1`, you can "roll back" your HEAD commit the same way we did before.

```
$ pachctl create branch data@master --head staging^1
```

This will kick off a new job to process `staging^1`. The HEAD commit on your output repo will be the result of processing `staging^1` instead of `staging`.

47.2 More complicated staging patterns

Using a `staging` branch allows you to defer processing, but it's inflexible, in that you need to know ahead of time what you want your input commits to look like. Sometimes you want to be able to commit data in an ad-hoc, disorganized way and then organize it later. For this, instead of updating your `master` branch to point at commits from `staging`, you can copy files directly from `staging` to `master`. With `copy file`, this only copies references, it doesn't move the actual data for the files around.

This would look like:

```
$ pachctl start commit data@master
$ pachctl copy file data@staging:file1 data@master
$ pachctl copy file data@staging:file2 data@master
...
$ pachctl finish commit data@master
```

You can also, of course, issue `delete file` and `put file` while the commit is open if you want to remove something from the parent commit or add something that isn't stored anywhere else.

47.3 Deferred processing on pipeline outputs

So far this document has focussed on deferred processing of data in input repos, however the same techniques apply to output repos. The only real difference is that rather than committing to a `staging` branch, you tell your pipeline to commit to that branch, by setting the `output_branch` field in your pipeline spec. Then when you want to process data you'd do:

```
$ pachctl create-branch pipeline master --head staging
```

CHAPTER 48

Vault Secret Engine

Pachyderm supports Vault integration by providing a Vault Secret Engine.

48.1 Deployment

Vault instructions for the admin deploying/configuring/managing vault

1. Get plugin binary

- Navigate to the Pachyderm Repo on github
  - Go to the latest release page
  - Download the `vault` asset

1. Download / Install that binary on your vault server instance

On your vault server:

```
# Assuming the binary was downloaded to /tmp/vault-plugins/pachyderm
export SHASUM=$(shasum -a 256 "/tmp/vault-plugins/pachyderm" | cut -d " " -f1)
echo $SHASUM
vault write sys/plugins/catalog/pachyderm sha_256="$SHASUM" command="pachyderm"
vault secrets enable -path=pachyderm -plugin-name=pachyderm plugin
```

Note: You may need to enable memory locking on the pachyderm plugin (see [https://www.vaultproject.io/docs/configuration/#disable_mlock]). That will look like:

```
sudo setcap cap_ipc_lock=+ep $(readlink -f /tmp/vault-plugins/pachyderm)
```

1. Configure the plugin

We'll need to gather and provide this information to the plugin for it to work:

- `admin_token` : is the (machine user) pachyderm token the plugin will use to cut new credentials on behalf of users
- `pachd_address` : is the URL where the pachyderm cluster can be accessed
- `ttl` : is the max TTL a token can be issued

48.1.1 Admin Token

To get a machine user `admin_token` from pachyderm:

If auth is not activated

(this activates auth with a robot user. It's also possible to activate auth with a github user. Also, the choice of `robot:admin` is arbitrary. You could name this admin `robot:<any string>`)

```
$ pachctl auth activate --initial-admin=robot:admin
Retrieving Pachyderm token...
WARNING: DO NOT LOSE THE ROBOT TOKEN BELOW WITHOUT ADDING OTHER ADMINS.
IF YOU DO, YOU WILL BE PERMANENTLY LOCKED OUT OF YOUR CLUSTER!
Pachyderm token for "robot:admin":
34cffc9254df40f0a277ee23e9fb005d

$ ADMIN_TOKEN=34cffc9254df40f0a277ee23e9fb005d
$ echo "${ADMIN_TOKEN}" | pachctl auth use-auth-token # authenticates you as the␣
↪cluster admin
```

If auth *is* already activated

```
# Login as a cluster admin
$ pachctl auth login
... login as cluster admin ...

# Appoint a new robot user as the cluster admin (if needed)
$ pachctl auth modify-admins --add=robot:admin

# Get a token for that robot user admin
$ pachctl auth get-auth-token robot:admin
New credentials:
  Subject: robot:admin
  Token: 3090e53de6cb4108a2c6591f3cbd4680

$ ADMIN_TOKEN=3090e53de6cb4108a2c6591f3cbd4680
```

Pass the new admin token to Pachyderm:

```
vault write pachyderm/config \
    admin_token="${ADMIN_TOKEN}" \
    pachd_address="${PACHD_ADDRESS:-127.0.0.1:30650}" \
    ttl=5m # optional
```

1. Test the plugin

```
vault read pachyderm/version

# If this fails, check if the problem is in the client (rather than the server):
vault read pachyderm/version/client-only
```

1. Manage user tokens with `revoke`

```
$ vault token revoke d2f1f95c-2445-65ab-6a8b-546825e4997a
Success! Revoked token (if it existed)
```

Which will revoke the vault token. But if you also want to manually revoke a pachyderm token, you can do so by issuing:

```
$vault write pachyderm/revoke user_token=xxx
```

48.2 Usage

When your application needs to access pachyderm, you will first do the following:

1. Connect / login to vault

Depending on your language / deployment this can vary. see the vault documentation for more details.

1. Anytime you are going to issue a request to a pachyderm cluster first:

- check to see if you have a valid pachyderm token
  - if you do not have a token, hit the `login` path as described below
  - if you have a token but it's TTL will expire soon (latter half of TTL is what's recommended), hit the `renew` path as described below
- then use the response token when constructing your client to talk to the pachyderm cluster

48.2.1 Login

Again, your client could be in any language. But as an example using the vault CLI:

```
$ vault write -f pachyderm/login/robot:test
Key                   Value
---                   -----
lease_id              pachyderm/login/robot:test/e93d9420-7788-4846-7d1a-8ac4815e4274
lease_duration        768h
lease_renewable       true
pachd_address         192.168.99.100:30650
user_token            aa425375f03d4a5bb0f529379d82aa39
```

The response metadata contains the `user_token` that you need to use to connect to the pachyderm cluster, as well as the `pachd_address`. Again, if you wanted to use this Pachyderm token on the command line:

```
$ echo "aa425375f03d4a5bb0f529379d82aa39" | pachctl auth use-auth-token
$ PACHD_ADDRESS=127.0.0.1:30650 pachctl list repo
```

The TTL is tied to the vault lease in `lease_id`, which can be inspected or revoked using the vault lease API (documented here: https://www.vaultproject.io/api/system/leases.html):

```
$ vault write /sys/leases/lookup lease_id=pachyderm/login/robot:test/e93d9420-7788-
↪4846-7d1a-8ac4815e4274
Key                Value
---                -----
expire_time        2018-06-17T23:32:23.317795215-07:00
id                 pachyderm/login/robot:test/e93d9420-7788-4846-7d1a-8ac4815e4274
issue_time         2018-05-16T23:32:23.317794929-07:00
last_renewal       <nil>
renewable          true
ttl                2764665
```

48.2.2 Renew

You should issue a `renew` request once the halfway mark of the TTL has elapsed. Like revocation, renewal is handled using the vault lease API:

```
$ vault write /sys/leases/renew lease_id=pachyderm/login/robot:test/e93d9420-7788-
↪4846-7d1a-8ac4815e4274 increment=3600
Key                Value
---                -----
lease_id           pachyderm/login/robot:test/e93d9420-7788-4846-7d1a-8ac4815e4274
lease_duration     2h
lease_renewable    true
```

CHAPTER 49

Pipeline Specification

This document discusses each of the fields present in a pipeline specification. To see how to use a pipeline spec to create a pipeline, refer to the pachctl create pipeline doc.

49.1 JSON Manifest Format

```
{
  "pipeline": {
    "name": string
  },
  "description": string,
  "transform": {
    "image": string,
    "cmd": [ string ],
    "stdin": [ string ],
    "err_cmd": [ string ],
    "err_stdin": [ string ],
    "env": {
        string: string
    },
    "secrets": [ {
        "name": string,
        "mount_path": string
    },
    {
        "name": string,
        "env_var": string,
        "key": string
    } ],
    "image_pull_secrets": [ string ],
    "accept_return_code": [ int ],
    "debug": bool,
    "user": string,
    "working_dir": string,
  },
  "parallelism_spec": {
    // Set at most one of the following:
    "constant": int,
    "coefficient": number
  },
  "hashtree_spec": {
   "constant": int,
  },
  "resource_requests": {
    "memory": string,
    "cpu": number,
    "disk": string,
  },
  "resource_limits": {
    "memory": string,
    "cpu": number,
    "gpu": {
      "type": string,
      "number": int
    }
    "disk": string,
  },
  "datum_timeout": string,
  "datum_tries": int,
  "job_timeout": string,
  "input": {
    <"pfs", "cross", "union", "cron", or "git" see below>
  },
  "output_branch": string,
  "egress": {
    "URL": "s3://bucket/dir"
  },
  "standby": bool,
  "cache_size": string,
  "enable_stats": bool,
  "service": {
    "internal_port": int,
    "external_port": int
  },
  "max_queue_size": int,
  "chunk_spec": {
    "number": int,
    "size_bytes": int
  },
  "scheduling_spec": {
    "node_selector": {string: string},
    "priority_class_name": string
  },
  "pod_spec": string,
  "pod_patch": string,
}

------------------------------------
"pfs" input
------------------------------------

"pfs": {
  "name": string,
  "repo": string,
  "branch": string,
  "glob": string,
  "lazy" bool,
  "empty_files": bool
}
```

```
------------------------------------
"cross" or "union" input
------------------------------------

"cross" or "union": [
  {
    "pfs": {
      "name": string,
      "repo": string,
      "branch": string,
      "glob": string,
      "lazy" bool,
      "empty_files": bool
    }
  },
  {
    "pfs": {
      "name": string,
      "repo": string,
      "branch": string,
      "glob": string,
      "lazy" bool,
      "empty_files": bool
    }
  }
  etc...
]

------------------------------------
"cron" input
------------------------------------

"cron": {
    "name": string,
    "spec": string,
    "repo": string,
    "start": time,
    "overwrite": bool
}

------------------------------------
"git" input
------------------------------------

"git": {
  "URL": string,
  "name": string,
  "branch": string
}
```

In practice, you rarely need to specify all the fields. Most fields either come with sensible defaults or can be empty. The following text is an example of a minimum spec:

```
{
  "pipeline": {
    "name": "wordcount"
  },
  "transform": {
    "image": "wordcount-image",
    "cmd": ["/binary", "/pfs/data", "/pfs/out"]
  },
  "input": {
        "pfs": {
            "repo": "data",
            "glob": "/*"
        }
    }
}
```

49.1.1 Name (required)

`pipeline.name` is the name of the pipeline that you are creating. Each pipeline needs to have a unique name. Pipeline names must meet the following prerequisites:

- Include only alphanumeric characters, `_` and `-` .
- Begin or end with only alphanumeric characters (not `_` or `-`).
- Not exceed 50 characters in length.

49.1.2 Description (optional)

`description` is an optional text field where you can add information about the pipeline.

49.1.3 Transform (required)

`transform.image` is the name of the Docker image that your jobs use.

`transform.cmd` is the command passed to the Docker run invocation. Similarly to Docker, `cmd` is not run inside a shell which means that wildcard globbing (`*`), pipes (`|`), and file redirects (`>` and `>>`) do not work. To specify these settings, you can set `cmd` to be a shell of your choice, such as `sh` and pass a shell script to `stdin` .

`transform.stdin` is an array of lines that are sent to your command on `stdin` . Lines do not have to end in newline characters.

`transform.err_cmd` is an optional command that is executed on failed datums. If the `err_cmd` is successful and returns 0 error code, it does not prevent the job from succeeding. This behavior means that `transform.err_cmd` can be used to ignore failed datums while still writing successful datums to the output repo, instead of failing the whole job when some datums fail. The `transform.err_cmd` command has the same limitations as `transform.cmd` .

`transform.err_stdin` is an array of lines that are sent to your error command on `stdin` . Lines do not have to end in newline characters.

`transform.env` is a key-value map of environment variables that Pachyderm injects into the container.

Note: There are environment variables that are automatically injected into the container, for a comprehensive list of them see the *Environment Variables* section below.

`transform.secrets` is an array of secrets. You can use the secrets to embed sensitive data, such as credentials. The secrets reference Kubernetes secrets by name and specify a path to map the secrets or an environment variable (`env_var`) that the value should be bound to. Secrets must set `name` which should be the name of a secret in

Kubernetes. Secrets must also specify either `mount_path` or `env_var` and `key`. See more information about Kubernetes secrets here.

`transform.image_pull_secrets` is an array of image pull secrets, image pull secrets are similar to secrets except that they are mounted before the containers are created so they can be used to provide credentials for image pulling. For example, if you are using a private Docker registry for your images, you can specify it by running the following command:

```
$ kubectl create secret docker-registry myregistrykey --docker-server=DOCKER_REGISTRY_
↪SERVER --docker-username=DOCKER_USER --docker-password=DOCKER_PASSWORD --docker-
↪email=DOCKER_EMAIL
```

And then, notify your pipeline about it by using `"image_pull_secrets": [ "myregistrykey" ]`. Read more about image pull secrets here.

`transform.accept_return_code` is an array of return codes, such as exit codes from your Docker command that are considered acceptable. If your Docker command exits with one of the codes in this array, it is considered a successful run to set job status. `0` is always considered a successful exit code.

`transform.debug` turns on added debug logging for the pipeline.

`transform.user` sets the user that your code runs as, this can also be accomplished with a `USER` directive in your `Dockerfile`.

`transform.working_dir` sets the directory that your command runs from. You can also specify the `WORKDIR` directive in your `Dockerfile`.

`transform.dockerfile` is the path to the `Dockerfile` used with the `--build` flag. This defaults to `./Dockerfile`.

49.1.4 Parallelism Spec (optional)

`parallelism_spec` describes how Pachyderm parallelizes your pipeline. Currently, Pachyderm has two parallelism strategies: `constant` and `coefficient`.

If you set the `constant` field, Pachyderm starts the number of workers that you specify. For example, set `"constant":10` to use 10 workers.

If you set the `coefficient` field, Pachyderm starts a number of workers that is a multiple of your Kubernetes cluster's size. For example, if your Kubernetes cluster has 10 nodes, and you set `"coefficient": 0.5`, Pachyderm starts five workers. If you set it to 2.0, Pachyderm starts 20 workers (two per Kubernetes node).

The default if left unset is "constant=1".

49.1.5 Resource Requests (optional)

`resource_requests` describes the amount of resources you expect the workers for a given pipeline to consume. Knowing this in advance lets Pachyderm schedule big jobs on separate machines, so that they do not conflict and either slow down or die.

The `memory` field is a string that describes the amount of memory, in bytes, each worker needs (with allowed SI suffixes (M, K, G, Mi, Ki, Gi, and so on). For example, a worker that needs to read a 1GB file into memory might set `"memory": "1.2G"` with a little extra for the code to use in addition to the file. Workers for this pipeline will be placed on machines with at least 1.2GB of free memory, and other large workers will be prevented from using it (if they also set their `resource_requests`).

The `cpu` field is a number that describes the amount of CPU time in `cpu seconds/real seconds` that each worker needs. Setting `"cpu": 0.5` indicates that the worker should get 500ms of CPU time per second. Setting `"cpu": 2` indicates that the worker gets 2000ms of CPU time per second. In other words, it is using 2 CPUs, though worker threads might spend 500ms on four physical CPUs instead of one second on two physical CPUs.

The `disk` field is a string that describes the amount of ephemeral disk space, in bytes, each worker needs with allowed SI suffixes (M, K, G, Mi, Ki, Gi, and so on).

In both cases, the resource requests are not upper bounds. If the worker uses more memory than it is requested, it does not mean that it will be shut down. However, if the whole node runs out of memory, Kubernetes starts deleting pods that have been placed on it and exceeded their memory request, to reclaim memory. To prevent deletion of your worker node, you must set your `memory` request to a sufficiently large value. However, if the total memory requested by all workers in the system is too large, Kubernetes cannot schedule new workers because no machine has enough unclaimed memory. `cpu` works similarly, but for CPU time.

By default, workers are scheduled with an effective resource request of 0 (to avoid scheduling problems that prevent users from being unable to run pipelines). This means that if a node runs out of memory, any such worker might be terminated.

For more information about resource requests and limits see the Kubernetes docs on the subject.

49.1.6 Resource Limits (optional)

`resource_limits` describes the upper threshold of allowed resources a given worker can consume. If a worker exceeds this value, it will be evicted.

The `gpu` field is a number that describes how many GPUs each worker needs. Only whole number are supported, Kubernetes does not allow multiplexing of GPUs. Unlike the other resource fields, GPUs only have meaning in Limits, by requesting a GPU the worker will have sole access to that GPU while it is running. It's recommended to enable `standby` if you are using GPUs so other processes in the cluster will have access to the GPUs while the pipeline has nothing to process. For more information about scheduling GPUs see the Kubernetes docs on the subject.

49.1.7 Datum Timeout (optional)

`datum_timeout` is a string (e.g. `1s`, `5m`, or `15h`) that determines the maximum execution time allowed per datum. So no matter what your parallelism or number of datums, no single datum is allowed to exceed this value.

49.1.8 Datum Tries (optional)

`datum_tries` is a int (e.g. `1`, `2`, or `3`) that determines the number of retries that a job should attempt given failure was observed. Only failed datums are retries in retry attempt. The the operation succeeds in retry attempts then job is successful, otherwise the job is marked as failure.

49.1.9 Job Timeout (optional)

`job_timeout` is a string (e.g. `1s`, `5m`, or `15h`) that determines the maximum execution time allowed for a job. It differs from `datum_timeout` in that the limit gets applied across all workers and all datums. That means that you'll need to keep in mind the parallelism, total number of datums, and execution time per datum when setting this value. Keep in mind that the number of datums may change over jobs. Some new commits may have a bunch of new files (and so new datums). Some may have fewer.

49.1.10 Input (required)

`input` specifies repos that will be visible to the jobs during runtime. Commits to these repos will automatically trigger the pipeline to create new jobs to process them. Input is a recursive type, there are multiple different kinds of inputs which can be combined together. The `input` object is a container for the different input types with a field for each, only one of these fields be set for any instantiation of the object.

```
{
    "pfs": pfs_input,
    "union": union_input,
    "cross": cross_input,
    "cron": cron_input
}
```

PFS Input

Note: Atom inputs were renamed to PFS inputs in version 1.8.1. If you are using an older version of Pachyderm, replace every instance of `pfs` with `atom` in the code below.

PFS inputs are the simplest inputs, they take input from a single branch on a single repo.

```
{
    "name": string,
    "repo": string,
    "branch": string,
    "glob": string,
    "lazy" bool,
    "empty_files": bool
}
```

`input.pfs.name` is the name of the input. An input with the name `XXX` is visible under the path `/pfs/XXX` when a job runs. Input names must be unique if the inputs are crossed, but they may be duplicated between `PFSInput` s that are combined by using the `union` operator. This is because when `PFSInput` s are combined, you only ever see a datum from one input at a time. Overlapping the names of combined inputs allows you to write simpler code since you no longer need to consider which input directory a particular datum comes from. If an input's name is not specified, it defaults to the name of the repo. Therefore, if you have two crossed inputs from the same repo, you must give at least one of them a unique name.

`input.pfs.repo` is the `repo` to be used for the input.

`input.pfs.branch` is the `branch` to watch for commits. If left blank, `master` is used by default.

`input.pfs.glob` is a glob pattern that is used to determine how the input data is partitioned. It is explained in detail in the next section.

`input.pfs.lazy` controls how the data is exposed to jobs. The default is `false` which means the job eagerly downloads the data it needs to process and exposes it as normal files on disk. If lazy is set to `true` , data is exposed as named pipes instead, and no data is downloaded until the job opens the pipe and reads it. If the pipe is never opened, then no data is downloaded.

Some applications do not work with pipes. For example, pipes do not support applications that makes `syscalls` such as `Seek` . Applications that can work with pipes must use them since they are more performant. The difference will be especially notable if the job only reads a subset of the files that are available to it.

Note: `lazy` currently does not support datums that contain more than 10000 files.

`input.pfs.empty_files` controls how files are exposed to jobs. If set to `true` , it causes files from this PFS to be presented as empty files. This is useful in shuffle pipelines where you want to read the names of files and reorganize them by using symlinks.

Union Input

Union inputs take the union of other inputs. In the example below, each input includes individual datums, such as if `foo` and `bar` were in the same repository with the glob pattern set to `/*`. Alternatively, each of these datums might have come from separate repositories with the glob pattern set to `/` and being the only filesystm objects in these repositories.

```
| inputA | inputB | inputA  inputB |
| ------ | ------ | --------------- |
| foo    | fizz   | foo             |
| bar    | buzz   | fizz            |
|        |        | bar             |
|        |        | buzz            |
```

The union inputs do not take a name and maintain the names of the sub-inputs. In the example above, you would see files under `/pfs/inputA/...` or `/pfs/inputB/...`, but never both at the same time. When you write code to address this behavior, make sure that your code first determines which input directory is present. Starting with Pachyderm 1.5.3, we recommend that you give your inputs the same `Name`. That way your code only needs to handle data being present in that directory. This only works if your code does not need to be aware of which of the underlying inputs the data comes from.

`input.union` is an array of inputs to combine. The inputs do not have to be `pfs` inputs. They can also be `union` and `cross` inputs. Although, there is no reason to take a union of unions because union is associative.

Cross Input

Cross inputs create the cross product of other inputs. In other words, a cross input creates tuples of the datums in the inputs. In the example below, each input includes individual datums, such as if `foo` and `bar` were in the same repository with the glob pattern set to `/*`. Alternatively, each of these datums might have come from separate repositories with the glob pattern set to `/` and being the only filesystm objects in these repositories.

```
| inputA | inputB | inputA  inputB |
| ------ | ------ | --------------- |
| foo    | fizz   | (foo, fizz)     |
| bar    | buzz   | (foo, buzz)     |
|        |        | (bar, fizz)     |
|        |        | (bar, buzz)     |
```

The cross inputs above do not take a name and maintain the names of the sub-inputs. In the example above, you would see files under `/pfs/inputA/...` and `/pfs/inputB/...`.

`input.cross` is an array of inputs to cross. The inputs do not have to be `pfs` inputs. They can also be `union` and `cross` inputs. Although, there is no reason to take a union of unions because union is associative.

Cron Input

Cron inputs allow you to trigger pipelines based on time. A Cron input is based on the Unix utility called `cron`. When you create a pipeline with one or more Cron inputs, `pachd` creates a repo for each of them. The start time for Cron input is specified in its spec. When a Cron input triggers, `pachd` commits a single file, named by the current RFC 3339 timestamp to the repo which contains the time which satisfied the spec.

```
{
    "name": string,
    "spec": string,
    "repo": string,
    "start": time,
    "overwrite": bool
}
```

`input.cron.name` is the name for the input. Its semantics is similar to those of `input.pfs.name`. Except that it is not optional.

`input.cron.spec` is a cron expression which specifies the schedule on which to trigger the pipeline. To learn more about how to write schedules, see the Wikipedia page on cron. Pachyderm supports non-standard schedules, such as `"@daily"`.

`input.cron.repo` is the repo which Pachyderm creates for the input. This parameter is optional. If you do not specify this parameter, then `"<pipeline-name>_<input-name>"` is used by default.

`input.cron.start` is the time to start counting from for the input. This parameter is optional. If you do not specify this parameter, then the time when the pipeline was created is used by default. Specifying a time enables you to run on matching times from the past or skip times from the present and only start running on matching times in the future. Format the time value according to RFC 3339.

`input.cron.overwrite` is a flag to specify whether you want the timestamp file to be overwritten on each tick. This parameter is optional, and if you do not specify it, it defaults to simply writing new files on each tick. By default, `pachd` expects only the new information to be written out on each tick and combines that data with the data from the previous ticks. If `"overwrite"` is set to `true`, it expects the full dataset to be written out for each tick and replaces previous outputs with the new data written out.

Git Input (alpha feature)

Git inputs allow you to pull code from a public git URL and execute that code as part of your pipeline. A pipeline with a Git Input will get triggered (i.e. will see a new input commit and will spawn a job) whenever you commit to your git repository.

Note: This only works on cloud deployments, not local clusters.

`input.git.URL` must be a URL of the form: `https://github.com/foo/bar.git`

`input.git.name` is the name for the input, its semantics are similar to those of `input.pfs.name`. It is optional.

`input.git.branch` is the name of the git branch to use as input

Git inputs also require some additional configuration. In order for new commits on your git repository to correspond to new commits on the Pachyderm Git Input repo, we need to setup a git webhook. At the moment, only GitHub is supported. (Though if you ask nicely, we can add support for GitLab or BitBucket).

1. Create your Pachyderm pipeline with the Git Input.
2. To get the URL of the webhook to your cluster, do `pachctl inspect pipeline` on your pipeline. You should see a `Githook URL` field with a URL set. Note - this will only work if you've deployed to a cloud provider (e.g. AWS, GKE). If you see `pending` as the value (and you've deployed on a cloud provider), it's possible that the service is still being provisioned. You can check `kubectl get svc` to make sure you see the `githook` service running.
3. To setup the GitHub webhook, navigate to:

```
https://github.com/<your_org>/<your_repo>/settings/hooks/new
```

Or navigate to webhooks under settings. Then you'll want to copy the `Githook URL` into the 'Payload URL' field.

49.1.11 Output Branch (optional)

This is the branch where the pipeline outputs new commits. By default, it's "master".

49.1.12 Egress (optional)

`egress` allows you to push the results of a Pipeline to an external data store such as s3, Google Cloud Storage or Azure Storage. Data will be pushed after the user code has finished running but before the job is marked as successful.

49.1.13 Standby (optional)

`standby` indicates that the pipeline should be put into "standby" when there's no data for it to process. A pipeline in standby will have no pods running and thus will consume no resources, it's state will be displayed as "standby".

Standby replaces `scale_down_threshold` from releases prior to 1.7.1.

49.1.14 Cache Size (optional)

`cache_size` controls how much cache a pipeline's sidecar containers use. In general, your pipeline's performance will increase with the cache size, but only up to a certain point depending on your workload.

Every worker in every pipeline has a limited-functionality `pachd` server running adjacent to it, which proxies PFS reads and writes (this prevents thundering herds when jobs start and end, which is when all of a pipeline's workers are reading from and writing to PFS simultaneously). Part of what these "sidecar" pachd servers do is cache PFS reads. If a pipeline has a cross input, and a worker is downloading the same datum from one branch of the input repeatedly, then the cache can speed up processing significantly.

49.1.15 Enable Stats (optional)

`enable_stats` turns on stat tracking for the pipeline. This will cause the pipeline to commit to a second branch in its output repo called `"stats"`. This branch will have information about each datum that is processed including: timing information, size information, logs and a `/pfs` snapshot. This information can be accessed through the `inspect datum` and `list datum` pachctl commands and through the webUI.

Note: enabling stats will use extra storage for logs and timing information. However it will not use as much extra storage as it appears to due to the fact that snapshots of the `/pfs` directory, which are generally the largest thing stored, don't actually require extra storage because the data is already stored in the input repos.

49.1.16 Service (alpha feature, optional)

`service` specifies that the pipeline should be treated as a long running service rather than a data transformation. This means that `transform.cmd` is not expected to exit, if it does it will be restarted. Furthermore, the service will be exposed outside the container using a kubernetes service. `"internal_port"` should be a port that the user code binds to inside the container, `"external_port"` is the port on which it is exposed, via the NodePorts functionality of kubernetes services. After a service has been created you should be able to access it at `http://<kubernetes-host>:<external_port>`.

49.1.17 Max Queue Size (optional)

`max_queue_size` specifies that maximum number of datums that a worker should hold in its processing queue at a given time (after processing its entire queue, a worker "checkpoints" its progress by writing to persistent storage). The default value is `1` which means workers will only hold onto the value that they're currently processing.

Increasing this value can improve pipeline performance, as that allows workers to simultaneously download, process and upload different datums at the same time (and reduces the total time spent on checkpointing). Decreasing this value can make jobs more robust to failed workers, as work gets checkpointed more often, and a failing worker will not lose as much progress. Setting this value too high can also cause problems if you have `lazy` inputs, as there's a cap of 10,000 `lazy` files per worker and multiple datums that are running all count against this limit.

49.1.18 Chunk Spec (optional)

`chunk_spec` specifies how a pipeline should chunk its datums.

`chunk_spec.number` if nonzero, specifies that each chunk should contain `number` datums. Chunks may contain fewer if the total number of datums don't divide evenly.

`chunk_spec.size_bytes` , if nonzero, specifies a target size for each chunk of datums. Chunks may be larger or smaller than `size_bytes` , but will usually be pretty close to `size_bytes` in size.

49.1.19 Scheduling Spec (optional)

`scheduling_spec` specifies how the pods for a pipeline should be scheduled.

`scheduling_spec.node_selector` allows you to select which nodes your pipeline will run on. Refer to the Kubernetes docs on node selectors for more information about how this works.

`scheduling_spec.priority_class_name` allows you to select the prioriy class for the pipeline, which will how Kubernetes chooses to schedule and deschedule the pipeline. Refer to the Kubernetes docs on priority and preemption for more information about how this works.

49.1.20 Pod Spec (optional)

`pod_spec` is an advanced option that allows you to set fields in the pod spec that haven't been explicitly exposed in the rest of the pipeline spec. A good way to figure out what JSON you should pass is to create a pod in Kubernetes with the proper settings, then do:

```
kubectl get po/<pod-name> -o json | jq .spec
```

this will give you a correctly formated piece of JSON, you should then remove the extraneous fields that Kubernetes injects or that can be set else where.

The JSON is applied after the other parameters for the `pod_spec` have already been set as a JSON Merge Patch. This means that you can modify things such as the storage and user containers.

49.1.21 Pod Patch (optional)

`pod_patch` is similar to `pod_spec` above but is applied as a JSON Patch. Note, this means that the process outlined above of modifying an existing pod spec and then manually blanking unchanged fields won't work, you'll need to create a correctly formatted patch by diffing the two pod specs.

49.2 The Input Glob Pattern

Each PFS input needs to specify a glob pattern.

Pachyderm uses the glob pattern to determine how many "datums" an input consists of. Datums are the unit of parallelism in Pachyderm. That is, Pachyderm attempts to process datums in parallel whenever possible.

Intuitively, you may think of the input repo as a file system, and you are applying the glob pattern to the root of the file system. The files and directories that match the glob pattern are considered datums.

For instance, let's say your input repo has the following structure:

```
/foo-1
/foo-2
/bar
  /bar-1
  /bar-2
```

Now let's consider what the following glob patterns would match respectively:

- `/` : this pattern matches `/` , the root directory itself, meaning all the data would be a single large datum.
- `/*` : this pattern matches everything under the root directory given us 3 datums: `/foo-1.` , `/foo-2.` , and everything under the directory `/bar` .
- `/bar/*` : this pattern matches files only under the `/bar` directory: `/bar-1` and `/bar-2`
- `/foo*` : this pattern matches files under the root directory that start with the characters `foo`
- `/*/*` : this pattern matches everything that's two levels deep relative to the root: `/bar/bar-1` and `/bar/bar-2`

The datums are defined as whichever files or directories match by the glob pattern. For instance, if we used `/*` , then the job will process three datums (potentially in parallel): `/foo-1` , `/foo-2` , and `/bar` . Both the `bar-1` and `bar-2` files within the directory `bar` would be grouped together and always processed by the same worker.

49.3 PPS Mounts and File Access

49.3.1 Mount Paths

The root mount point is at `/pfs` , which contains:

- `/pfs/input_name` which is where you would find the datum.
  - Each input will be found here by its name, which defaults to the repo name if not specified.
- `/pfs/out` which is where you write any output.

Environment Variables

There are several environment variables that get injected into the user code before it runs. They are:

- `PACH_JOB_ID` the id the currently run job.
- `PACH_OUTPUT_COMMIT_ID` the id of the commit being outputted to.
- For each input there will be an environment variable with the same name defined to the path of the file for that input. For example if you are accessing an input called `foo` from the path `/pfs/foo` which contains a file called `bar` then the environment variable `foo` will have the value `/pfs/foo/bar` . The path in the environment variable is the path which matched the glob pattern, even if the file is a directory, ie if your glob pattern is `/*` it would match a directory `/bar` , the value of `$foo` would then be `/pfs/foo/bar` . With a glob pattern of `/*/*` you would match the files contained in `/bar` and thus the value of `foo` would be `/pfs/foo/bar/quux` .
- For each input there will be an environment variable named `input_COMMIT` indicating the id of the commit being used for that input.

In addition to these environment variables Kubernetes also injects others for Services that are running inside the cluster. These allow you to connect to those outside services, which can be powerful but also can be hard to reason about, as processing might be retried multiple times. For example if your code writes a row to a database that row may be written multiple times due to retries. Interaction with outside services should be idempotent to prevent unexpected behavior. Furthermore, one of the running services that your code can connect to is Pachyderm itself, this is generally not recommended as very little of the Pachyderm API is idempotent, but in some specific cases it can be a viable approach.

CHAPTER 51

Pachctl Command Line Tool

Pachctl is the command line interface for Pachyderm. To install Pachctl, follow the *Local Installation* instructions

51.1 Synopsis

Access the Pachyderm API.

Environment variables:

PACHD_ADDRESS=<host>:<port>, the pachd server to connect to (e.g. 127.0.0.1:30650).

51.2 Options

--no-metrics	Don't report user metrics for this command
-v, --verbose	Output verbose logs

CHAPTER 52

Pachyderm language clients

52.1 Go Client

The Go client is officially supported by the Pachyderm team. It implements almost all of the functionality that is provided with the `pachctl` CLI tool, and, thus, you can easily integrated operations like `put file` into your applications.

For more info, check out the godocs.

Note - A compatible version of `grpc` is needed when using the Go client. You can deduce the compatible version from our vendor.json file, where you will see something like:

```
        {
            "checksumSHA1": "mEyChIkG797MtkrJQXW8X/qZ0l0=",
            "path": "google.golang.org/grpc",
            "revision": "21f8ed309495401e6fd79b3a9fd549582aed1b4c",
            "revisionTime": "2017-01-27T15:26:01Z"
        },
```

You can then get this version via:

```
go get google.golang.org/grpc
cd $GOPATH/src/google.golang.org/grpc
git checkout 21f8ed309495401e6fd79b3a9fd549582aed1b4c
```

52.2 Python Client -

The Python client is a user contributed client that has just recently been brought under the Pachyderm umbrella and made into an official client. We're working on getting it fully up to date and will be supporting it full going forward.

For more info, check out `pypachy` on GitHub.

52.3 Scala Client

Our users are currently working on a Scala client for Pachyderm. Please contact us if you are interested in helping with this or testing it out.

52.4 Other languages

Pachyderm uses a simple protocol buffer API. Protobufs support a bunch of other languages, any of which can be used to programmatically use Pachyderm. We haven't built clients for them yet, but it's not too hard. It's an easy way to contribute to Pachyderm if you're looking to get involved.

S3Gateway API

This outlines the HTTP API exposed by the s3gateway and any peculiarities relative to S3. The operations largely mirror those documented in S3's official docs.

Generally, you would not call these endpoints directly, but rather use a tool or library designed to work with S3-like APIs. Because of that, some working knowledge of S3 and HTTP is assumed.

53.1 Operations on buckets

Buckets are represented via `branch.repo`, e.g. the `master.images` bucket corresponds to the `master` branch of the `images` repo.

53.1.1 Creating buckets

Route: `PUT /branch.repo/`.

If the repo does not exist, it is created. If the branch does not exist, it is likewise created. As per S3's behavior in some regions (but not all), trying to create the same bucket twice will return a `BucketAlreadyOwnedByYou` error.

53.1.2 Deleting buckets

Route: `DELETE /branch.repo/`.

Deletes the branch. If it is the last branch in the repo, the repo is also deleted. Unlike S3, you can delete non-empty branches.

53.1.3 Listing buckets

Route: `GET /`.

Lists all of the branches across all of the repos as S3 buckets.

53.2 Object operations

Object operations act upon the HEAD commit of branches. Authorization-gated PFS branches are not supported.

53.2.1 Writing objects

Route: `PUT /branch.repo/filepath`.

Writes the PFS file at `filepath` in an atomic commit on the HEAD of `branch`. Any existing file content is overwritten. Unlike S3, there is no limit to upload size.

The s3gateway does not support multipart uploads, but you can use this endpoint to upload very large files. We recommend setting the `Content-MD5` request header - especially for larger files - to ensure data integrity.

Some S3 libraries and clients will detect that our s3gateway does not support multipart uploads and automatically fallback to using this endpoint. Notably, this includes minio.

53.2.2 Removing objects

Route: `DELETE /branch.repo/filepath`.

Deletes the PFS file `filepath` in an atomic commit on the HEAD of `branch`.

53.2.3 Listing objects

Route: `GET /branch.repo/`

Only S3's list objects v1 is supported.

PFS directories are represented via `CommonPrefixes`. This largely mirrors how S3 is used in practice, but leads to a couple of differences:

- If you set the delimiter parameter, it must be `/`.
- Empty directories are included in listed results.

With regard to listed results:

- Due to PFS peculiarities, the `LastModified` field references when the most recent commit to the branch happened, which may or may not have modified the specific object listed.
- The HTTP `ETag` field does not use MD5, but is a cryptographically secure hash of the file contents.
- The S3 `StorageClass` and `Owner` fields always have the same filler value.

53.2.4 Getting objects

Route: `GET /branch.repo/filepath`.

There is support for range queries and conditional requests, however error response bodies for bad requests using these headers are not standard S3 XML.

With regard to HTTP response headers:

- Due to PFS peculiarities, the HTTP `Last-Modified` header references when the most recent commit to the branch happened, which may or may not have modified this specific object.
- The HTTP `ETag` does not use MD5, but is a cryptographically secure hash of the file contents.