Pachyderm Documentation

Release 1.1.0

Joe Doliner

Jul 26, 2017

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.

CHAPTER 1

Getting Started

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

Local Installation: Get Pachyderm deployed locally on OSX or Linux.

Beginner Tutorial: Learn to use Pachdyerm through a quick and simple tutorial.

troubleshooting: Common getting started issues and how to fix them.

If you'd like to read about the technical concepts in Pachyderm before actually running it, check out our reference docs:

- ../reference/pachyderm_file_system
- ../reference/pachyderm_pipeline_system
- *Pachctl Command Line Tool*
- Use Cases

—

If you've already got a Kubernetes cluster running or would rather use AWS, GCE or Azure, check out our *Intro*.

—

Looking for in-depth development docs?

Learn how to *Creating Analysis Pipelines* check out more advanced Pachyderm examples such as word count or machine learning with TensorFlow.

CHAPTER 2

Local Installation

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

If you hit any errors not covered in this guide, check our 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.

Prerequisites

- *Minikube* (and VirtualBox)
- *Pachyderm Command Line Interface*

Minikube

Kubernetes offers a fantastic guide to install minikube. Follow the Kubernetes installation guide to install Virtual Box, Minikibe, and Kubectl. Then come back here to install Pachyderm.

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.

Pachctl

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

```
# For OSX:
$ brew tap pachyderm/tap && brew install pachyderm/tap/pachctl@1.5

# For Linux (64 bit):
$ curl -o /tmp/pachctl.deb -L https://github.com/pachyderm/pachyderm/releases/
↪download/v1.5.0/pachctl_1.5.0_amd64.deb && sudo dpkg -i /tmp/pachctl.deb
```

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.

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 pachd nodes to be running because it's pulling containers from DockerHub. You can see the cluster status by using `kubectl get all`:

```
$ kubectl get all
NAME                         READY     STATUS    RESTARTS   AGE
po/etcd-4197107720-br61m     1/1       Running   0          8m
po/pachd-3548222380-s086m    1/1       Running   2          8m

NAME              CLUSTER-IP     EXTERNAL-IP   PORT(S)                         AGE
svc/etcd          10.111.11.36   <nodes>       2379:32379/TCP                  8m
svc/kubernetes    10.96.0.1      <none>        443/TCP                         10m
svc/pachd         10.97.116.5    <nodes>       650:30650/TCP,651:30651/TCP     8m

NAME           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/etcd    1         1         1            1           8m
deploy/pachd   1         1         1            1           8m

NAME                    DESIRED   CURRENT   READY     AGE
rs/etcd-4197107720      1         1         1         8m
rs/pachd-3548222380     1         1         1         8m
```

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

Port Forwarding

The last step is to set up port forwarding so commands you send can reach Pachyderm within the VM. We background this process since port forwarding blocks.

```
$ pachctl port-forward &
```

Once port forwarding is complete, pachctl should automatically be connected. Try `pachctl version` to make sure everything is working.

```
$ pachctl version
COMPONENT           VERSION
pachctl             1.4.6
pachd               1.4.6
```

We're good to go!

If for any reason `port-forward` doesn't work, you can connect directly by setting `ADDRESS` to the minikube IP with port 30650.

```
$ minikube ip
192.168.99.100
$ export ADDRESS=192.168.99.100:30650
```

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 analysis pipelines.

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 you'll be introduced to the basic concepts of Pachyderm.

Image processing with OpenCV

In this guide we're going to create a Pachyderm pipeline to do some simple edge detection on a few images. Thanks to Pachyderm's processing system, we'll be able to run the pipeline in a distributed, streaming fashion. 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, check our 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.

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.

Create a Repo

A `repo` is the highest level data primitive in Pachyderm. Like many things in Pachyderm, it shares it's name with primitives 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

# See the repo we just created
$ pachctl list-repo
NAME                CREATED             SIZE
images              2 minutes ago       0 B
```

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 two flags, `-c` and `-f`. `-f` can take either a local file or a URL which it'll automatically scrape. In our case, we'll simply pass the URL.

Unlike Git though, 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. The `-c` flag specifies that we want to start a new commit, add data, and finish the commit in a convenient one-liner.

We also specify the repo name "images", the branch name "master", and what we want to name the file, "liberty.png".

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

Finally, we 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
images              5 minutes ago   57.27 KiB

# We can view the commit we just created
$ pachctl list-commit images
REPO                ID                                 PARENT              STARTED
↪         DURATION            SIZE
images              7162f5301e494ec8820012576476326c   <none>              2 minutes
↪ago      38 seconds          57.27 KiB

# And view the file in that commit
$ pachctl list-file images master
NAME                TYPE                SIZE
liberty.png         file                57.27 KiB
```

We can 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.

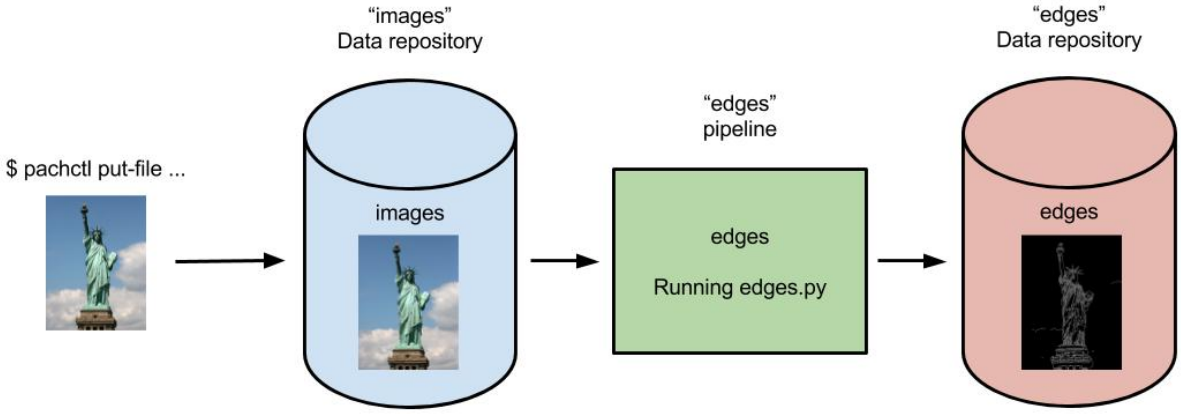
```
# on OSX
$ pachctl get-file images master liberty.png | open -f -a /Applications/Preview.app

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

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. This includes 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": {
    "atom": {
      "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 "atom" input, our images repo with a particular glob pattern.

The glob pattern defines how the input data can be broken up if we wanted 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))
```

Our python code is really straight forward. We're simply walking 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 for you. All the input data for a pipeline will be found in `/pfs/[input_repo_name]` and your code should always write to `/pfs/out` .

Now let's create the pipeline in Pachyderm:

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

What Happens When You Create a Pipeline

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

This first time it runs a pipeline it needs to download the image from DockerHub so this might take a minute. Every subsequent run will be much faster.

You can view the job with:

```
$ pachctl list-job
ID                                   OUTPUT COMMIT                                  ␣
↪STARTED            DURATION           STATE
a6c70aa5-9f0c-4e36-b30a-4387fac54eac edges/1a9c76a2cd154e6e90f200fb80c46d2f   2␣
↪minutes ago      About a minute     success
```

Every pipeline creates a corresponding repo with the same name where it stores its output results. In our example, the "edges" pipeline created a repo called "edges" to store the results.

```
$ pachctl list-repo
NAME                CREATED             SIZE
edges               2 minutes ago       22.22 KiB
images              10 minutes ago      57.27 KiB
```

Reading the Output

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

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

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

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 with `-c` 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 -c -f http://imgur.com/8MN9Kg0.png

$ pachctl put-file images master kitten.png -c -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                                   OUTPUT COMMIT                                ␣
↪STARTED            DURATION         STATE
7395c7c9-df0e-4ea8-8202-ec846970b982 edges/8848e11056c04518a8d128b6939d9985  2␣
↪minutes ago      Less than a second  success
b90afeb1-c12b-4ca5-a4f4-50c50efb20bb edges/da51395708cb4812bc8695bb151b69e3  2␣
↪minutes ago      1 seconds           success
9182d65e-ea36-4b98-bb07-ebf40fefcce5 edges/4dd2459531414d80936814b13b1a3442  5␣
↪minutes ago      3 seconds           success
```

```
# View the output data

# on OSX
$ 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
```

Exploring the File System (optional)

Another nifty feature of Pachyderm is that you can mount the file system locally to poke around and explore your data using FUSE. FUSE comes pre-installed on most Linux distributions. For OS X, you'll need to install OSX FUSE. This is just an optional step if you want another view of your data and system and can be useful for local development.

The first thing we need to do is mount Pachyderm's filesystem (pfs).

First create the mount point:

```
$ mkdir ~/pfs
```

And then mount it:

```
# We background this process because it blocks.
$ pachctl mount ~/pfs &
```

Note: If you get any errors on OSX, those are most likely benign as it's just SpotLight trying to index the Fuse volume and not having access.

This will mount pfs on `~/pfs` you can inspect the filesystem like you would any other local filesystem such as using `ls` or pointing your browser at it.

Note: Use `pachctl unmount ~/pfs` to unmount the filesystem. You can also use the `-a` flag to remove all Pachyderm FUSE mounts.

Next Steps

We've now got Pachyderm running locally with data and a pipeline! If you want to keep playing with Pachyderm locally, you can use what you've learned to build on or change this pipeline. You can also start learning some of the more advanced topics to develop analysis in Pachyderm:

- *Intro*
- *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 buid our version control semantics and are therefore is not "human-readable" directly in the object store. That being said, Pachyderm allows you and your pipeline stages to interact with versioned files like you would in a normal file system.

Jargon associated with putting data in Pachyderm

"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.

"Commits"

Regardless of the method you use to get data into Pachyderm, the mechanism that is used to get data into Pachyderm 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"). Then the data put into Pachyderm in that open commit will only be available once the commit is "finished" (aka a "closed commit"). Although you have to do this opening, putting, and closing for all data that is committed into Pachyderm, we provide some convient ways to do that with our CLI tool and clients (see below).

How to get data into Pachyderm

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

- The `pachctl` CLI tool: This is the great option for testing and for users who prefer to input data scripting.
- 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 writtern 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.

`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. Note again, commits in Pachyderm must be explicitly started and finished so `put-file` can only be called on an open commit (started, but not finished). The `-c` option allows you to start and finish a commit in addition to putting data as a one-line command.

Add a single file to a new branch:

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

# then utilize the returned <commit-id> in the put-file request
# to put <file> at <path> in the <repo>
$ pachctl put-file <repo> <commit-id> </path/to/file> -f <file>

# then finish the commit
$ pachctl finish-commit <repo> <commit-id>
```

Start and finish a commit while adding a file using `-c`:

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

Put data from a URL:

```
$ pachctl put-file <repo> <branch> </path/to/file> -c -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> -c -f s3://object_store_url
```

Put data directly from another location within Pachyderm:

```
$ pachctl put-file <repo> <branch> </path/to/file> -c -f pfs://pachyderm_location
```

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> -c -i <file containing list of files, paths, or␣
↪URLs>
```

Pipe data from stdin into a data repository:

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

Add an entire directory by using the recursive flag, `-r`:

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

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.

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. For more tips on composing pipelines see "Composing Pipelines".

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 :doc: `parallelization` to see the details of that.

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.

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, your Docker image should NOT specifiy a `CMD` . Rather, you specify what commands are to be run in the container when you create your pipeline.

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 `--push-images` flag on `update-pipeline` to help you tag your images as they are updated. See the updating pipelines docs for more information.

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": {
      "atom": {
        "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 *every* finished commit on 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.

Note - In Pachyderm 1.4+, as soon as you create your pipeline, Pachyderm will launch worker pods on Kubernetes, such that they are ready to process any data committed to their input repos.

CHAPTER 6

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?*

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.

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.

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 `atom` 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": {
    "atom": {
        "repo": "string",
        "glob": "string",
    }
}
```

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

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

More information about "atoms," unions, and crosses can be found in our Pipeline Specification.

Datums

Pachyderm uses the glob pattern to determine how many "datums" an input atom 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.

Defining Datums via Glob Patterns

Intuitively, you should think of the input atom 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 input atom 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 it's own datum.

Glob patterns also let you take only a particular directory (or subset of directories) as an input atom 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.

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 input atom 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 it's 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.

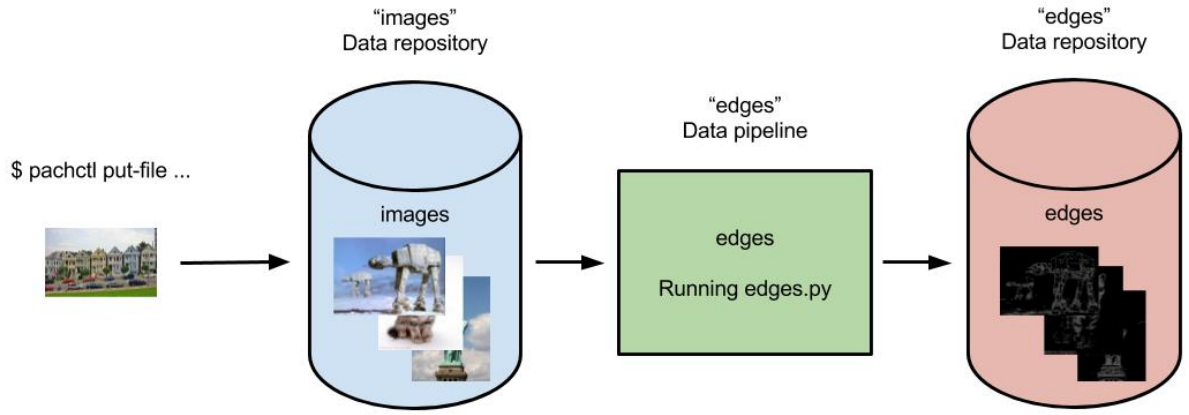
If our glob pattern was `/*/*` then each `<city>.json` file would be it's own datum. That means if we added a `Sacramento.json` file, only that specific file would be processed by Pachyderm.

CHAPTER 7

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.

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> <commit-id or branch> 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.

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 input atoms, 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 commmit:

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

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.

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

Although `pachctl` and `output` 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).

CHAPTER 8

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.

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.

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 piplines. To update the code in your pipeline:

1. Make the code changes.
2. Re-build your Docker image.
3. Call `update-pipeline` with the `--push-images` flag.

You need to call `update-pipeline` with the `--push-images` flag because, if you have already run your pipeline, Pachyderm has already pulled the specified images. It won't re-pull new versions of the images, unless we tell it to (which ensures that we don't waste time pulling images when we don't need to). When `--push-images` is specified, Pachyderm will do the following:

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

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

```
pachctl update-pipeline -f edges.json --push-images --password <registry password> -u
↪<registry user>
```

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 reprocess everything from scratch, you should use the `--reprocess` flag. This will reprocess all previously processed data and all new data with the new code. Previous results will still be available in pfs.

CHAPTER 9

Examples

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

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

Machine Learning

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 commponents, where the pipeline includes the trianing 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

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

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

CHAPTER 10

Intro

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 docs cover the following commonly used deployments:

- Google Cloud Platform
- Amazon Web Services
- Azure
- OpenShift
- On Premises
- Custom Object Stores
- Migrations

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.

CHAPTER 11

Google Cloud Platform

Google Cloud Platform has excellent support for Kubernetes through the Google Container Engine.

Prerequisites

- Google Cloud SDK >= 124.0.0
- kubectl

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

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

```
$ gcloud components install kubectl
```

This will download the `kubectl` binary to `google-cloud-sdk/bin`

Deploy Kubernetes

To create a new Kubernetes cluster in 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 for the k8s nodes. We recommend "n1-standard-4" or larger.]

# By default this 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}
```

Note that 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.

This may take a few minutes to start up. You can check the status on the GCP Console. Then, after the cluster is up, you can point `kubectl` to this cluster via:

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

As a sanity check, make sure your cluster is up and running via `kubectl`:

```
$ kubectl get all
NAME             CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
svc/kubernetes   10.0.0.1     <none>        443/TCP   22s
```

Deploy Pachyderm

To deploy Pachyderm we will need to:

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

Set up the Storage Resources

Pachyderm needs a GCS bucket and a persistent disk to function correctly. The create the persistent 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
↪"]

# Name this whatever you want, we chose pach-disk as a default
$ STORAGE_NAME=pach-disk

$ gcloud compute disks create --size=${STORAGE_SIZE}GB ${STORAGE_NAME}
```

Then we need to specify the bucket name and create the bucket:

```
# 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:

```
$ gcloud compute instances list
# should see a number of instances

$ gsutil ls
# should see a bucket

$ gcloud compute disks list
# should see a number of disks, including the one you specified
```

Install `pachctl`

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

```
# For OSX:
$ brew tap pachyderm/tap && brew install pachyderm/tap/pachctl@1.5

# For Linux (64 bit):
$ curl -o /tmp/pachctl.deb -L https://github.com/pachyderm/pachyderm/releases/
↪download/v1.5.0/pachctl_1.5.0_amd64.deb && sudo dpkg -i /tmp/pachctl.deb
```

You can try running `pachctl version` to check that this worked correctly, but Pachyderm itself isn't deployed yet so you won't get a `pachd` version.

```
$ pachctl version
COMPONENT           VERSION
pachctl             1.4.6
pachd               (version unknown) : error connecting to pachd server at address␣
↪(0.0.0.0:30650): context deadline exceeded

please make sure pachd is up (`kubectl get all`) and portforwarding is enabled
```

Deploy Pachyderm

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

```
pachctl deploy google ${BUCKET_NAME} ${STORAGE_SIZE} --static-etcd-volume=${STORAGE_
↪NAME} --dashboard
```

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 by using:

```
$ kubectl get all
NAME                         READY     STATUS    RESTARTS   AGE
po/etcd-4197107720-br61m     1/1       Running   0          8m
po/pachd-3548222380-s086m    1/1       Running   2          8m

NAME              CLUSTER-IP     EXTERNAL-IP   PORT(S)                         AGE
svc/etcd          10.111.11.36   <nodes>       2379:32379/TCP                  8m
svc/kubernetes    10.96.0.1      <none>        443/TCP                         10m
svc/pachd         10.97.116.5    <nodes>       650:30650/TCP,651:30651/TCP     8m

NAME           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/etcd    1         1         1            1           8m
deploy/pachd   1         1         1            1           8m

NAME                    DESIRED   CURRENT   READY     AGE
rs/etcd-4197107720      1         1         1         8m
rs/pachd-3548222380     1         1         1         8m
```

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 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             1.4.6
pachd               1.4.6
```

CHAPTER 12

Amazon Web Services

Below, we show how to deploy Pachyderm on AWS in a couple of different ways:

1. By manually deploying Kubernetes and Pachyderm.
2. By executing a one shot deploy script that will both deploy Kubernetes and Pachyderm.

If you already have a Kubernetes deployment or would like to customize the types of instances, size of volumes, etc. in your Kubernetes cluster, you should follow option (1). If you just want a quick deploy to experiment with Pachyderm in AWS or would just like to use our default configuration, you might want to try option (2)

Production Deployment

Note - for production deployments we recommend setting up AWS CloudFront. AWS puts S3 rate limits in place that can limit the data throughput for your cluster, and CloudFront helps mitigate this issue.

Follow the instructions here to deploy a Pachyderm cluster with CloudFront

Prerequisites

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

Manual Pachyderm Deploy

Deploy Kubernetes

The easiest way to install Kubernetes on AWS is with kops. Kubenetes has provided a step by step guide for the deploy. Please follow this guide to deploy Kubernetes on AWS.

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             CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
svc/kubernetes   10.0.0.1     <none>        443/TCP   22s
```

Deploy Pachyderm

To deploy Pachyderm we 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`, a command-line utility used for Pachyderm. To install `pachctl` run one of the following:

```
# For OSX:
$ brew tap pachyderm/tap && brew install pachyderm/tap/pachctl@1.5

# For Linux (64 bit):
$ curl -o /tmp/pachctl.deb -L https://github.com/pachyderm/pachyderm/releases/
↪download/v1.5.0/pachctl_1.5.0_amd64.deb && sudo dpkg -i /tmp/pachctl.deb
```

You can try running `pachctl version` to check that this worked correctly, but Pachyderm itself isn't deployed yet so you won't get a `pachd` version.

```
$ pachctl version
COMPONENT           VERSION
pachctl             1.4.6
pachd               (version unknown) : error connecting to pachd server at address␣
↪(0.0.0.0:30650): context deadline exceeded.
```

Set up the Storage Resources

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

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

```
$ kubectl cluster-info
  Kubernetes master is running at https://1.2.3.4
  ...
$ KUBECTLFLAGS="-s [The public IP of the Kubernetes master. e.g. 1.2.3.4]"

# 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)]

$ AWS_AVAILABILITY_ZONE=[the AWS availability zone of your Kubernetes cluster. e.g.
↪"us-west-2a"]
```

Then to actually create the resources, you can run:

```
$ aws s3api create-bucket --bucket ${BUCKET_NAME} --region ${AWS_REGION} --create-
↪bucket-configuration LocationConstraint=${AWS_REGION}

$ aws ec2 create-volume --size ${STORAGE_SIZE} --region ${AWS_REGION} --availability-
↪zone ${AWS_AVAILABILITY_ZONE} --volume-type gp2
```

Record the "volume-id" that is output (e.g. "vol-8050b807") from the above `create-volume` command as shown below (you can also view it in the aws console or with `aws ec2 describe-volumes`):

```
$ STORAGE_NAME=<volume id>
```

Now, as a sanity check, you should be able to see the bucket and the EBS volume that are just created:

```
aws s3api list-buckets --query 'Buckets[].Name'
aws ec2 describe-volumes --query 'Volumes[].VolumeId'
```

Deploy Pachyderm

When you installed kops, you should have created a dedicated IAM user (see here for details). To deploy Pachyderm you will need to export these credentials to the following environmental variables:

```
$ 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_ACCESS_KEY_ID} ${AWS_SECRET_ACCESS_KEY}
↪" " ${AWS_REGION} ${STORAGE_SIZE} --static-etcd-volume=${STORAGE_NAME}
```

(Note, the `" "` in the deploy command is for an optional temporary AWS token, if you are just experimenting with a deploy. Such a token should NOT be used for a production deploy). 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 by using:

```
$ kubectl get all
NAME                          READY     STATUS    RESTARTS   AGE
po/etcd-4197107720-br61m      1/1       Running   0          8m
po/pachd-3548222380-s086m     1/1       Running   2          8m

NAME              CLUSTER-IP     EXTERNAL-IP   PORT(S)                       AGE
svc/etcd          10.111.11.36   <nodes>       2379:32379/TCP                8m
svc/kubernetes    10.96.0.1      <none>        443/TCP                       10m
svc/pachd         10.97.116.5    <nodes>       650:30650/TCP,651:30651/TCP   8m

NAME           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/etcd    1         1         1            1           8m
deploy/pachd   1         1         1            1           8m

NAME                    DESIRED   CURRENT   READY     AGE
rs/etcd-4197107720      1         1         1         8m
rs/pachd-3548222380     1         1         1         8m
```

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, 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             1.4.6
pachd               1.4.6
```

One Shot Script

Install additional prerequisites

This scripted deploy requires a couple of prerequisites in addition to the ones listed under Prerequisites:

- jq
- uuid

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 all` should return something like:

```
NAME                          READY     STATUS    RESTARTS   AGE
po/etcd-4197107720-br61m      1/1       Running   0          8m
po/pachd-3548222380-s086m     1/1       Running   2          8m

NAME              CLUSTER-IP      EXTERNAL-IP   PORT(S)                         AGE
svc/etcd          10.111.11.36    <nodes>       2379:32379/TCP                  8m
svc/kubernetes    10.96.0.1       <none>        443/TCP                         10m
svc/pachd         10.97.116.5     <nodes>       650:30650/TCP,651:30651/TCP     8m

NAME            DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/etcd     1         1         1            1           8m
deploy/pachd    1         1         1            1           8m

NAME                     DESIRED   CURRENT   READY     AGE
rs/etcd-4197107720       1         1         1         8m
rs/pachd-3548222380      1         1         1         8m
```

Connect `pachctl`

Finally, 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`:

```
$ pachctl version
COMPONENT           VERSION
pachctl             1.4.6
pachd               1.4.6
```

CHAPTER 13

Azure

Prerequisites

- Install Azure CLI >= 2.0.1
- Install jq

Deploy Kubernetes

The easiest way to deploy a Kubernetes cluster is to use the official Kubernetes guide.

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.

Set up the Storage Resources

Pachyderm requires an object store (Azure Storage) and a data disk to function correctly.

Here are the parameters required to create these resources:

```
# 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]

$ CONTAINER_NAME=[The name of the Azure blob container 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'
$ az storage blob list \
  --container=${CONTAINER_NAME} \
  --account-name=${STORAGE_ACCOUNT} \
  --account-key=${STORAGE_KEY}
```

Install `pachctl`

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

```
# For OSX:
$ brew tap pachyderm/tap && brew install pachyderm/tap/pachctl@1.5

# For Linux (64 bit):
$ curl -o /tmp/pachctl.deb -L https://github.com/pachyderm/pachyderm/releases/
↪download/v1.5.0/pachctl_1.5.0_amd64.deb && sudo dpkg -i /tmp/pachctl.deb
```

You can try running `pachctl version` to check that this worked correctly, but Pachyderm itself isn't deployed

yet so you won't get a `pachd` version.

```
$ pachctl version
COMPONENT           VERSION
pachctl             1.4.6
pachd               (version unknown) : error connecting to pachd server at address␣
↪(0.0.0.0:30650): context deadline exceeded.
```

Deploy Pachyderm

Now we're ready to boot up Pachyderm:

```
$ pachctl deploy microsoft ${CONTAINER_NAME} ${STORAGE_ACCOUNT} ${STORAGE_KEY} $
↪{STORAGE_SIZE} --static-etcd-volume=${VOLUME_URI}
```

It may take a few minutes for the pachd nodes to be running because it's pulling containers from Docker Hub. You can see the cluster status by using:

```
NAME                         READY     STATUS    RESTARTS   AGE
po/etcd-4197107720-br61m     1/1       Running   0          8m
po/pachd-3548222380-s086m    1/1       Running   2          8m

NAME              CLUSTER-IP     EXTERNAL-IP   PORT(S)                         AGE
svc/etcd          10.111.11.36   <nodes>       2379:32379/TCP                  8m
svc/kubernetes    10.96.0.1      <none>        443/TCP                         10m
svc/pachd         10.97.116.5    <nodes>       650:30650/TCP,651:30651/TCP     8m

NAME            DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/etcd     1         1         1            1           8m
deploy/pachd    1         1         1            1           8m

NAME                    DESIRED   CURRENT   READY     AGE
rs/etcd-4197107720      1         1         1         8m
rs/pachd-3548222380     1         1         1         8m
```

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, 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             1.4.6
pachd               1.4.6
```

CHAPTER 14

OpenShift

OpenShift is a popular enterprise Kubernetes distribution. Pachyderm can run on OpenShift with some additional steps:

Deploy Pachyderm

1. Make sure that privilege containers are allowed (they are not allowed by default). You can add `priviledged` scc (SecurityContextConstraints) to `pachyderm` service account:

```
oadm policy add-scc-to-user privileged system:serviceaccount:<PROJECT_NAME>:pachyderm
```

or manually edit `oc edit scc privileged`:

```
users:
- system:serviceaccount:<PROJECT_NAME>:pachyderm
```

1. Replace `hostPath` with `emptyDir` in your cluster manifest (Your manifest is generated by the `pachctl deploy ...` command or can be generated manually. To only generate the manifest, run `pachctl deploy ...` with the `--dry-run` flag).

```
      "spec": {
        "volumes": [
          {
            "name": "pach-disk",
            "emptyDir": {}
          }
        ],

... <snip>  ...

      "spec": {
        "volumes": [
          {
            "name": "etcd-storage",
            "emptyDir": {}
          }
        ],
```

Please note that `emptyDir` does not persit your data. You need to configure persistent volume or hostPath to persist your data.

1. Deploy Pachyderm manifest you modified.

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

You can see the cluster status by using `oc get all` like kubernetes:

```
$ oc get all
NAME             DESIRED          CURRENT       AGE
rc/etcd          1                1             5m
rc/pachd         1                1             5m
NAME             CLUSTER-IP       EXTERNAL-IP   PORT(S)           AGE
svc/etcd         172.30.170.24    <nodes>       2379/TCP          5m
svc/pachd        172.30.194.202   <nodes>       650/TCP,651/TCP   5m
NAME             READY            STATUS        RESTARTS          AGE
po/etcd-7m5r1    1/1              Running       0                 5m
po/pachd-foq68   1/1              Running       0                 5m
```

Configure for a Pipeline

1. Add `cluster-reader` and `edit` role to `pachyderm` service account.

```
oadm policy add-cluster-role-to-user cluster-reader system:serviceaccount:<PROJECT_
↪NAME>:pachyderm
oadm policy add-cluster-role-to-user edit system:serviceaccount:<PROJECT_NAME>:
↪pachyderm
```

1. Add the `pachyderm` service account to the pipeline Pod (ReplicationController).

```
oc patch rc pipeline-edges-v1 -p 'spec:
  template:
    spec:
      serviceAccount: pachyderm
      serviceAccountName: pachyderm'
```

or manually edit rc `oc edit rc <RC_PIPELINE> -o json`:

```
                ...
              "dnsPolicy": "ClusterFirst",
              "serviceAccountName": "pachyderm",
              "serviceAccount": "pachyderm",
              "securityContext": {}
                ...
```

1. Replace `hostPath` with `emptyDir`.

Again, please note that `emptyDir` does not persit your data. You need to configure persistent volume or hostPath to persist.

1. Redeploy the updated Pods.

```
$ oc scale rc pipeline-edges-v1 --replicas=0
$ oc scale rc pipeline-edges-v1 --replicas=4
```

You can see the pipeline pods are running and successful job.

```
$ oc get pod
NAME                      READY     STATUS    RESTARTS   AGE
etcd-kbi4n                1/1       Running   0          1h
```

```
pachd-z3b7y                1/1       Running   0          1h
pipeline-edges-v1-28vdj    1/1       Running   0          12s
pipeline-edges-v1-fpa8v    1/1       Running   0          12s
pipeline-edges-v1-mshi0    1/1       Running   0          12s
pipeline-edges-v1-yx2wa    1/1       Running   0          12s

$ pachctl list-job
ID                                   OUTPUT COMMIT                           STARTED
↪      DURATION   RESTART PROGRESS STATE
1b2c1b49-f536-484f-b0e3-07b3906572be edges/006f0aecb2b048d5b5edee0cdb766879 55
↪minutes ago 51 minutes 0       1 / 1    success
```

Problems related to OpenShift deployment are tracked in this issue: https://github.com/pachyderm/pachyderm/issues/336. If you have additional related questions, please ask them on Pachyderm's slack channel or via email support@pachyderm.io.

CHAPTER 15

On Premises

Pachyderm is built on Kubernetes and can be backed by an object store of your choice. As such, Pachyderm can run on any on premise platforms/frameworks that support Kubernetes, a persistent disk/volume, and an object store.

Prerequisites

1. kubectl
2. pachctl

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.

Object Store

Once you have Kubernetes up and running, deploying Pachyderm is a matter of supplying Kubernetes with a JSON/yaml manifest to create the Pachyderm resources. This includes providing information that Pachyderm will use to connect to a backing object store.

For on premise deployments, we recommend using Minio as a backing object store. However, at this point, you could utilize any backing object store that has an S3 compatible API. To create a manifest template for your on premise deployment, run:

```
pachctl deploy custom --persistent-disk google --object-store s3 <persistent disk␣
↪name> <persistent disk size> <object store bucket> <object store id> <object store␣
↪secret> <object store endpoint> --static-etcd-volume=${STORAGE_NAME} --dry-run >␣
↪deployment.json
```

Then you can modify `deployment.json` to fit your environment and kubernetes deployment. Once, you have your manifest ready, deploying Pachyderm is as simple as:

```
kubectl create -f deployment.json
```

Need Help?

If you need help with your on premises deploy, please reach out to us on Pachyderm's slack channel or via email at support@pachyderm.io. We are happy to help!

CHAPTER 16

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.

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.

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}
```

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}
```

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 17

Migrations

Occationally, Pachyderm introduces changes that are backward-incompatible: repos/commits/files created on an old version of Pachyderm may be unusable on a new version of Pachyderm. When that happens, we try our best to write a migration script that "upgrades" your data so it's usable by the new version of Pachyderm.

Migrate to 1.4.x

To migrate to 1.4.x, look under the directory named `migration/X-Y`. For instance, to upgrade from 1.3.12 to 1.4.0, look under `migration/1.3.12-1.4.0`.

Note - If you are migrating from Pachyderm <= 1.3 to 1.4+, you should read this guide. In this particular case, a migration script is NOT provided due to significant changes in our processing and metadata structures.

Migrate to 1.5.x

To migrate from 1.4.x to 1.5.x, use the `pachctl migrate` command. See `pachctl migrate --help` for detailed instructions.

As an example, to migrate from 1.4.8 to 1.5.0, use the following command:

```
$ pachctl migrate --from 1.4.8 --to 1.5.0
```

Note that the `pachctl migrate` command can be run either before or after you've redeployed your cluster with the new version (e.g. via `pachctl deploy`).

Most importantly, you need to ensure that your cluster is "at rest" when you run `pachctl migrate`. That is, there shouldn't be any ongoing activities that are changing the state of the cluster. Examples would be running jobs or ongoing `put-file` requests.

Note: For v1.4 pipelines that specify environment variables in their pipeline specs, you will unfortunately need to reprocess all data for those pipelines as part of the v1.5 migration. This will automatically happen as part of the first job that spawns after the migration. Sorry for inconvenience.

Backup

It's paramount that you backup your data before running a migration. While we've tested the migration code extensively, it's still possible that they contain bugs, or that you accidentally use them in a wrong way.

In general, there are two data storage systems that you might consider backing up: the metadata storage and the data storage. Not all migration scripts touch both systems, so you might only need to back up one of them. Look at the README for a particular migration script for details.

Backup the metadata store

Assuming you've deployed Pachyderm on a public cloud, your metadata is probably stored on a persistent volume. See the respective Deploying Pachyderm guide for details.

Here are official guides on backing up persistent volumes for each cloud provider:

- GCE Persistent Volume
- Elastic Block Store (EBS)

Backup the object store

We don't currently have migration scripts that affect the object store.

CHAPTER 18

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).

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.

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.

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.

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_spec` 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.

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 proceeeds.

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 19

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.

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 `lazy input`. That way, your shuffling pipeline can skip both the download and the upload.

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.

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 accomodate 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.

CHAPTER 20

General Troubleshooting

Here are some common issues by symptom along with steps to resolve them. They are organized into the following categories:

- *Deploying a Pachyderm cluster*
- *Connecting to a Pachyderm cluster*
- *Problems running pipelines*

Deploying A Pachyderm Cluster

- *Pod stuck in `CrashLoopBackoff`*
- *Pod stuck in `CrashLoopBackoff` - with error attaching volume*

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 this one. 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)"
```

That 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 likely reveal a misconfiguration in your deploy. For example, you might see, `BucketRegionError: incorrect region,the bucket is not in 'us-west-2' region`. In that case, you've deployed your bucket in a different region than your cluster.

If the error / recourse isn't obvious from the error message, you can now provide the content of the `pachd` logs when getting help in our Slack channel or by opening a GitHub Issue. Please provide these logs either way as it is extremely helpful in resolving the issue..

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
```

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, first find the node of which the pod is scheduled. In the output of the `kubectl describe po/pachd-xxx` command above, you should see the name of the node on which the pod is running. In the AWS web console, find that node.. Once you have the right node, look in 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.

Connecting to a Pachyderm Cluster

Cannot connect via `pachctl` - context deadline exceeded

Symptom

You may be using the environmental variable `ADDRESS` to specify how `pachctl` talks to your Pachyderm cluster, or you may be forwarding the pachyderm port via `pachctl port-forward`. In any event, you might see something similar to:

```
$ echo $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).

Certificate Error When Using Kubectl

Symptom

This can happen on any request using `kubectl` (e.g. `kubectl get all`), but it can also be seen when running `pachctl port-forward` because it uses `kubectl` under the hood. 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
```

Uploads/Downloads are Slow

Symptom

Any `pachctl put-file` or `pachctl get-file` commands are slow.

Recourse

Check if you're using port-forwarding. Port forwarding throttles traffic to ~1MB/s. If you need to do large downloads/uploads you should consider using the `ADDRESS` variable instead to connect directly to your k8s master node. See this note

You'll also want to make sure you've allowed ingress access through any firewalls to your k8s cluster.

Problems Running Pipelines

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.

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 21

Deploy Specific Troubleshooting

Here are some common issues by symptom related to certain deploys. They are organized into the following categories:

- *AWS*
- Google - coming soon...
- Azure - coming soon...

AWS Deployment

- *Can't connec to to the Pachyderm cluster after a rolling update*
- *The one shot deploy script, `aws.sh`, never completes*
- *VPC limit exceeded*
- *GPU node never appears*

Can't connect to the Pachyderm cluster after a rolling update

Symptom

After running `kops rolling-update`, `kubectl` (and/or `pachctl`) cannot connect to the cluster. All `kubectl` requests hang.

Recourse

First get your cluster name. This will be in the deploy logs you saved from running `aws.sh` (if you utilized the one shot deployment), or can be retrieved via `kops get clusters`.

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
```

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 navigate there, open the AWS web console for EC2. Check to see if you have any instances with names like::

```
master-us-west-2a.masters.tfgpu.kubernetes.com
nodes.tfgpu.kubernetes.com
```

If not, 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.
```

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.

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 22

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 -c -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:

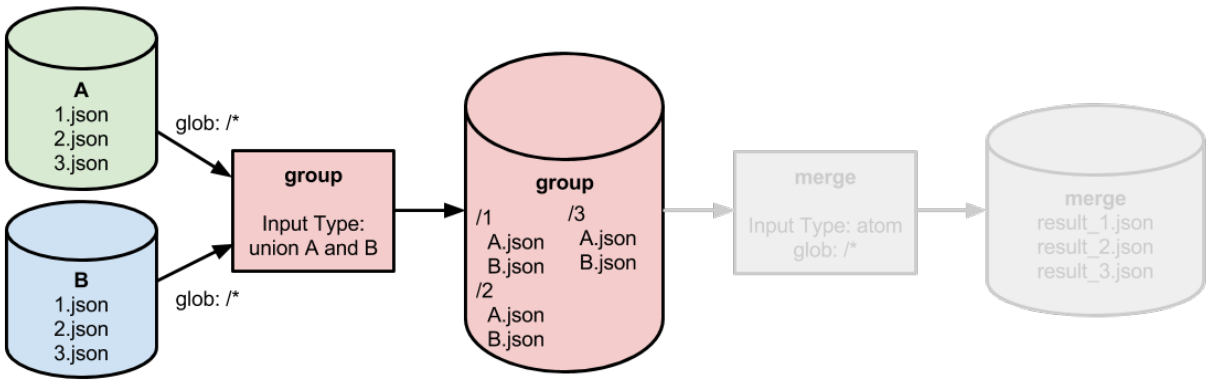
```
# Split a json file on json blobs, putting
# each json blob into it's own file.
$ pachctl put-file users master -c -f user_data.json --split json --target-file-
↪datums 1

# Split a json file on json blobs, putting
# 3 json blobs into each split file.
$ pachctl put-file users master -c -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 -c -f user_data.txt --split line --target-file-bytes␣
↪100
```

Combining or Merging 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.

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).

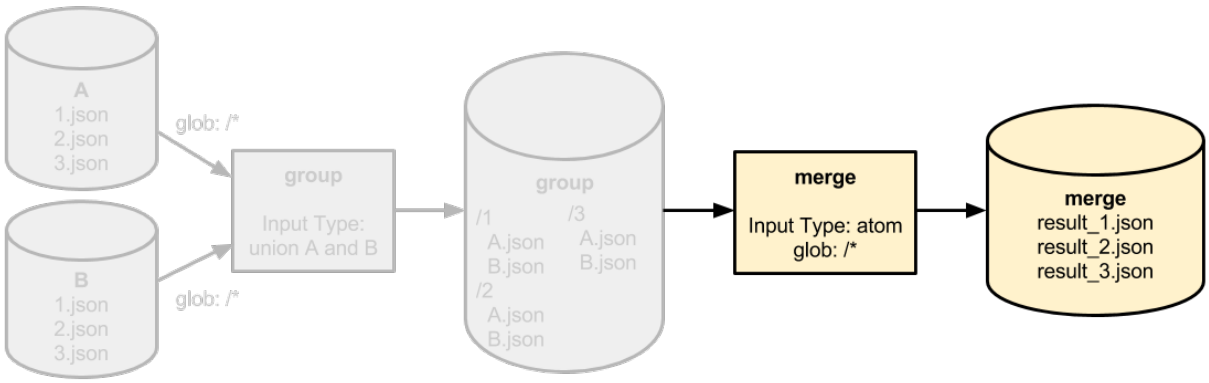
The first pipeline takes a union of `A` and `B` as inputs, each with glob pattern `/*`. As each JSON file is processed, it is copied to a folder in the output corresponding to the key/index for that record (in this case, just the number in the file name). It is also re-named to a unique name corresponding 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 unecessary downloads of data.
- You should use sym-links to avoid unecessary uploads of data and unecessary duplication of data (see more information on "copy elision" here).

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
```

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 24

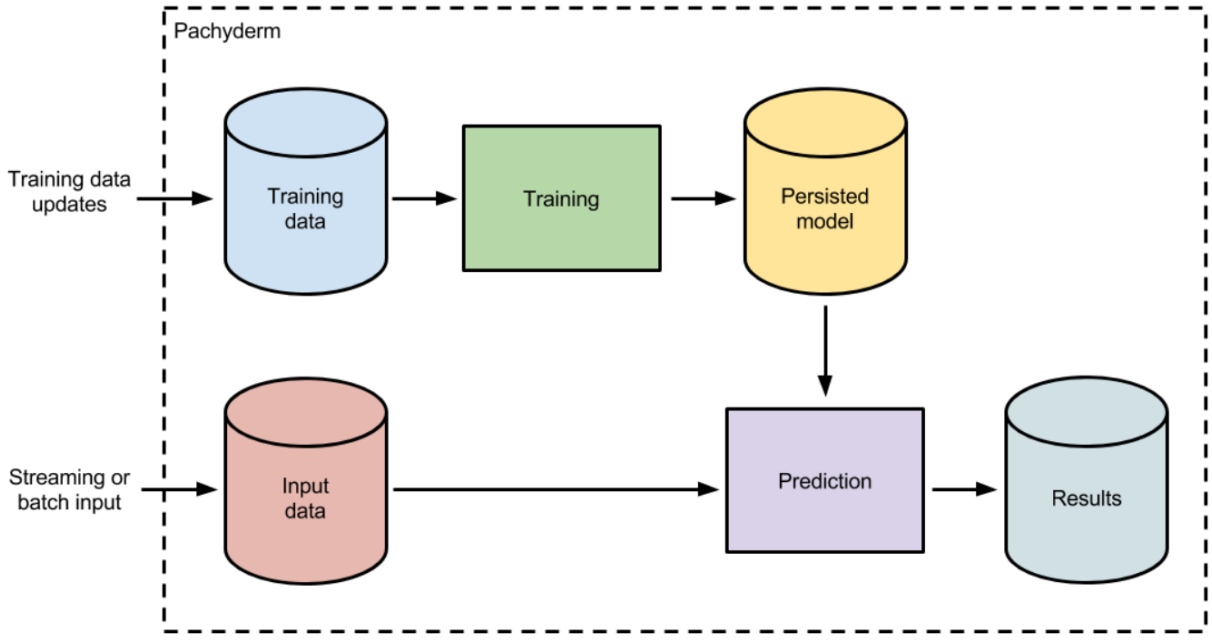
Creating Machine Learning Workflows

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!

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.

CHAPTER 25

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.

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.

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.

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
```

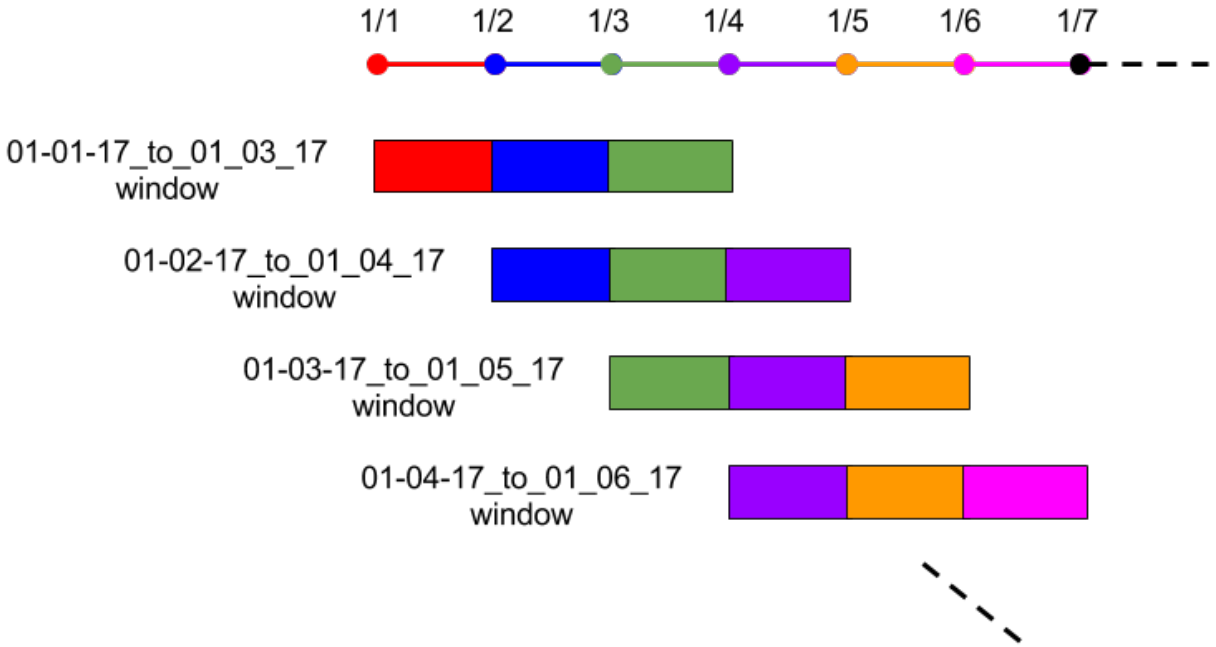
the first pipeline executes again to bin the 01-02-17 data into any relevant bins. In this case, we would put it in the previously created bin for 01-01-17 to 01-03-17, but we would also put it into a bin starting on 01-02-17:

```
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:



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 uneccessary 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 unecessary 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.

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 commited,

```
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.

CHAPTER 26

Utilizing GPUs

Pachyderm has alpha support for utilizing GPUs within Pachyderm pipelines (e.g., for training machine learning models). To do this you will need to:

1. Create a Docker image that is able to utilize GPUs
2. Write a pipeline spec that specifies GPU nodes
3. Deploy a GPU enabled pachyderm cluster

For a concrete example, see our example Tensorflow pipeline for image-to-image translation, which includes a pipeline specification for running model training on a GPU node.

Creating a GPU Enabled Docker Image

For your Docker image, you'll want to use or build an image that can utilize GPU resources. If you are using Tensorflow, for example, you could build your Docker image `FROM` the public GPU enabled Tensorflow image:

```
FROM tensorflow/tensorflow:0.12.0-gpu
...
```

Or you might follow this guide for working with Docker and NVIDIA (although we haven't full tested this guide).

Writing Your Pipeline Specification

Ensuring that your environment can access GPU drivers

You can bake this into your Docker image in some cases, but other images, such as the TensorFlow base image, may require that you explicitly tell your application about shared libraries (e.g., CUDA). To do that, you may need to set one or more environmental variables. This will be application/framework dependent. For example, if we were using the Tensorflow base image, we would need to make sure that we set `LD_LIBRARY_PATH` , such that TensorFlow knows about CUDA:

```
LD_LIBRARY_PATH="/usr/lib/nvidia:/usr/local/cuda/lib64:/rootfs/usr/lib/x86_64-linux-
↪gnu"
```

Again, this can be baked into your Docker image via an `ENV` statement in your Dockerfile:

```
ENV LD_LIBRARY_PATH /usr/lib/nvidia:/usr/local/cuda/lib64:/rootfs/usr/lib/x86_64-
↪linux-gnu
```

or it can be defined in your pipeline specification via the `env` field (as shown below).

Creating your pipeline specification with access to GPU resources

In addition to properly setting up the environment, we need to tell the Pachyderm cluster that our pipeline needs a GPU resource. To do that we'll add a `gpu` entry to the `resources` field in the pipeline specification.

An example pipeline definition for a GPU enabled Pachyderm Pipeline is as follows:

```
{
  "pipeline": {
    "name": "train"
  },
  "transform": {
    "image": "acme/your-gpu-image",
    "cmd": [
      "python",
      "train.py"
    ],
    "env": {
      "LD_LIBRARY_PATH": "/usr/lib/nvidia:/usr/local/cuda/lib64:/rootfs/usr/lib/x86_
↪64-linux-gnu"
    }
  },
  "resource_spec": {
      "gpu": 1
  },
  "inputs": {
    "atom": {
      "repo": "data",
      "glob": "/*"
    }
  ]
}
```

Deploy a GPU Enabled Pachyderm Cluster

NOTE: You can also *test Pachyderm + GPUs locally*

NOTE: The following has been tested with these versions of k8s/kops:

```
$kubectl version
Client Version: version.Info{Major:"1", Minor:"6", GitVersion:"v1.6.1", GitCommit:
↪"b0b7a323cc5a4a2019b2e9520c21c7830b7f708e", GitTreeState:"clean", BuildDate:"2017-
↪04-03T20:44:38Z", GoVersion:"go1.7.5", Compiler:"gc", Platform:"linux/amd64"}
Server Version: version.Info{Major:"1", Minor:"6", GitVersion:"v1.6.2", GitCommit:
↪"477efc3cbe6a7effca06bd1452fa356e2201e1ee", GitTreeState:"clean", BuildDate:"2017-
↪04-19T20:22:08Z", GoVersion:"go1.7.5", Compiler:"gc", Platform:"linux/amd64"}
$kops version
Version 1.6.0-beta.1 (git-77f222d)
```

To deploy a Pachyderm cluster with GPU support we assume:

- You're using kops for your deployment (which means you're using AWS or GCE, not GKE). Other deploy methods are available, but these are the ones we've tested most thoroughly.
- You have a working pachyderm cluster already up and running that you can connect to with `kubectl`.

Add GPU nodes to your k8s cluster

You can create GPU nodes by first using `kops` to create a new instance group:

```
$kops create ig gpunodes --name XXXXXX-pachydermcluster.kubernetes.com --state s3://
↪k8scom-state-store-pachyderm-YYYYY --subnet us-west-2c
```

where your specification will look something like:

```
 1 apiVersion: kops/v1alpha2
 2 kind: InstanceGroup
 3 metadata:
 4   creationTimestamp: null
 5   name: gpunodes
 6 spec:
 7   image: kope.io/k8s-1.5-debian-jessie-amd64-hvm-ebs-2017-01-09
 8   machineType: p2.xlarge
 9   maxSize: 1
10   minSize: 1
11   role: Node
12   subnets:
13   - us-west-2c
```

In this example we used Amazon's p2.xlarge instance which contains a single GPU node.

Node - If you upped your `rootVolumeSize` (and set the `rootVolumeType` in your other instance group), you should do the same here. In the absence of GPU jobs, normal jobs could get scheduled on this node, in which case you'll have the same disk requirements as the rest of your cluster. There is currently no way of setting "disk" resource requests, so we have to use a convention instead.

Enable GPUs at the k8s level

Again, you can use `kops` to edit your cluster:

```
$kops edit cluster --name XXXXXXX-pachydermcluster.kubernetes.com --state s3://k8scom-
↪state-store-pachyderm-YYYY
```

and add the fields:

```
  hooks:
  - execContainer:
      image: pachyderm/nvidia_driver_install:dcde76f919475a6585c9959b8ec41334b05103bb
  kubelet:
    featureGates:
      Accelerators: "true"
```

Note: It's YAML and spaces are very important. Also, if you see "fields were not recognized," you likely need to update the version of `kops`.

These lines provide an image that gets run on every node's startup. This image will install the NVIDIA drivers on the host machine, update the host machine to mount the device at startup, and restart the host machine.

The feature gate enables k8s GPU detection. That's what gives us the `alpha.kubernetes.io/nvidia-gypu: "1"` resources.

Update your cluster

Finally, we "update" our cluster to actually make the above changes:

```
$kops update cluster --name XXXXXXX-pachydermcluster.kubernetes.com --state s3://
↪k8scom-state-store-pachyderm-YYYY --yes
```

This will spin up the new `gpunodes` instance group, and apply the changes to your kops cluster.

Sanity check

You'll know the cluster is ready to schedule GPU resources when:

- you see the new node in the output of `kubectl get nodes` and the state is `Ready` , and
- the node has the `alpha.kubernetes.io/nvidia-gpu: "1"` field set (and the value is 1 not 0)

```
$kubectl get nodes/ip-172-20-38-179.us-west-2.compute.internal -o yaml | grep nvidia
    alpha.kubernetes.io/nvidia-gpu: "1"
    alpha.kubernetes.io/nvidia-gpu-name: Tesla-K80
    alpha.kubernetes.io/nvidia-gpu: "1"
    alpha.kubernetes.io/nvidia-gpu: "1"
```

Deal with known issues (if necessary)

If you're not seeing the node, its possible that your resource limits (from your cloud provider) are preventing you from creating the GPU node(s). You should check your resource limits and ensure that GPU nodes are available in your region/zone (as further discussed here).

If you have checked your resource limits and everything seems ok, its very possible that you're hitting a known k8s bug. In this case, you can try to overcome the issue by restarting the k8s api server. To do that, run:

```
kubectl --namespace=kube-system get pod | grep kube-apiserver | cut -f 1 -d " " |␣
↪while read pod; do kubectl --namespace=kube-system delete po/$pod; done
```

It can take a few minutes for the node to get recognized by the k8s cluster again.

Test Locally

NOTE - This has only been tested on a linux machine.

If you want to test that your pipeline is working on a local cluster (you're Pachyderm in a local cluster), you can do so, but you'll need to attach the NVIDIA drivers correctly. There are two methods for this:

1. Fresh install

Install the NVIDIA drivers locally if you haven't already. If you're not sure, run `which nvidia-smi` . If it returns no result, you probably don't have them installed. To install them, you can run the following command. Warning! This command will restart your system and will modify your `/etc/rc.local` file, which you may want to backup.

```
$sudo /usr/bin/docker run -v /:/rootfs/ -v /var/run/dbus:/var/run/dbus -v /run/
↪systemd:/run/systemd --net=host --privileged pachyderm/nvidia_driver_install:
↪dcde76f919475a6585c9959b8ec41334b05103bb
```

After the restart, you should see the nvidia devices mounted:

```
$ls /dev | grep nvidia
nvidia0
nvidiactl
nvidia-modeset
nvidia-uvm
```

At this point your local machine should be recognized by kubernetes. To check you'll do something like:

```
$kubectl get nodes
NAME        STATUS    AGE       VERSION
127.0.0.1   Ready     13d       v1.6.2
$kubectl get nodes/127.0.0.1 -o yaml | grep nvidia
    alpha.kubernetes.io/nvidia-gpu: "1"
    alpha.kubernetes.io/nvidia-gpu-name: Quadro-M2000M
    alpha.kubernetes.io/nvidia-gpu: "1"
    alpha.kubernetes.io/nvidia-gpu: "1"
```

If you don't see any `alpha.kubernetes.io/nvidia-gpu` fields it's likely that you didn't deploy k8s locally with the correct flags. An example of the right flags can be found here. You can clone git@github.com:pachyderm/pachyderm and run `make launch-kube` locally if you're already running docker on your local machine.

2. Hook in existing drivers

Pachyderm expects to find the shared libraries it needs under `/usr/lib`. It mounts in `/usr/lib` into the container as `/rootfs/usr/lib` (only when you've specified a GPU resource). In this case, if your drivers are not found, you can update the `LD_LIBRARY_PATH` in your container as appropriate.

CHAPTER 27

Pipeline Specification

This document discusses each of the fields present in a pipeline specification. To see how to use a pipeline spec, refer to the pachctl create-pipeline doc.

JSON Manifest Format

```
{
  "pipeline": {
    "name": string
  },
  "description": string,
  "transform": {
    "image": string,
    "cmd": [ string ],
    "stdin": [ string ]
    "env": {
        string: string
    },
    "secrets": [ {
        "name": string,
        "mount_path": string
    } ],
    "image_pull_secrets": [ string ],
    "accept_return_code": [ int ]
  },
  "parallelism_spec": {
    // Set at most one of the following:
    "constant": int
    "coefficient": double
  },
  "resource_spec": {
    "memory": string
    "cpu": double
  },
  "input": {
    <"atom" or "cross" or "union", see below>
  },
  "output_branch": string,
  "egress": {
    "URL": "s3://bucket/dir"
  },
```

```
  "scale_down_threshold": string,
  "incremental": bool,
  "cache_size": string
}

-------------------------------------
"atom" input
-------------------------------------

"atom": {
  "name": string,
  "repo": string,
  "branch": string,
  "glob": string,
  "lazy" bool,
  "from_commit": string
}

-------------------------------------
"cross" or "union" input
-------------------------------------

"cross" or "union": [
  {
    "atom": {
      "name": string,
      "repo": string,
      "branch": string,
      "glob": string,
      "lazy" bool,
      "from_commit": string
    }
  },
  {
    "atom": {
      "name": string,
      "repo": string,
      "branch": string,
      "glob": string,
      "lazy" bool,
      "from_commit": string
    }
  }
  etc...
]
```

In practice, you rarely need to specify all the fields. Most fields either come with sensible defaults or can be nil. Following is an example of a minimal spec:

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

```
  "input": {
        "atom": {
            "repo": "data",
            "glob": "/*"
        }
    }
}
```

Following is a walk-through of all the fields.

Name (required)

`pipeline.name` is the name of the pipeline that you are creating. Each pipeline needs to have a unique name.

Description (optional)

`description` is an optional text field where you can put documentation about the pipeline.

Transform (required)

`transform.image` is the name of the Docker image that your jobs run in.

`transform.cmd` is the command passed to the Docker run invocation. Note that as with Docker, cmd is not run inside a shell which means that things like wildcard globbing (`*`), pipes (`|`) and file redirects (`>` and `>>`) will not work. To get that behavior, you can set `cmd` to be a shell of your choice (e.g. `sh`) and pass a shell script to stdin.

`transform.stdin` is an array of lines that are sent to your command on stdin. Lines need not end in newline characters.

`transform.env` is a map from key to value of environment variables that will be injected into the container

`transform.secrets` is an array of secrets, secrets reference Kubernetes secrets by name and specify a path that the secrets should be mounted to. Secrets are useful for embedding sensitive data such as credentials. Read more about secrets in Kubernetes here.

`transform.image_pull_secrets` is an array of image pull secrets, image pull secrets are similar to secrets except that they're 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 via:

```
$ 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 tell your pipeline about it via `"image_pull_secrets": [ "myregistrykey" ]`. Read more about image pull secrets here.

`transform.accept_return_code` is an array of return codes (i.e. exit codes) from your docker command that are considered acceptable, which means that if your docker command exits with one of the codes in this array, it will be considered a successful run for the purpose of setting job status. `0` is always considered a successful exit code.

Parallelism Spec (optional)

`parallelism_spec` describes how Pachyderm should parallelize your pipeline. Currently, Pachyderm has two parallelism strategies: `constant` and `coefficient`.

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).

By default, we use the parallelism spec "coefficient=1", which means that we spawn one worker per node for this pipeline.

Resource Spec (optional)

`resource_spec` describes the amount of resources you expect the workers for a given pipeline to consume. Knowing this in advance lets us schedule big jobs on separate machines, so that they don't 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, etc). 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 only 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_spec`).

The `cpu` field is a double that describes the amount of CPU time (in (cpu seconds)/(real seconds) 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 should get 2000ms of CPU time per second (i.e. it's using 2 CPUs, essentially, though worker threads might spend e.g. 500ms on four physical CPUs instead of one second on two physical CPUs).

In both cases, the resource requests are not upper bounds. If the worker uses more memory than it's requested, it will not (necessarily) be killed. However, if the whole node runs out of memory, Kubernetes will start killing pods that have been placed on it and exceeded their memory request, to reclaim memory. To prevent your worker getting killed, 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 will be unable to schedule new workers (because no machine will have 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 killed.

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 insantiation of the object.

```
{
    "atom": atom_input,
    "union": [input],
    "cross": [input],
}
```

Atom Input

Atom 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,
    "from_commit": string
}
```

`input.atom.name` is the name of the input. An input with name `XXX` will be visible under the path `/pfs/XXX` when a job runs. Input names must be unique. If an input's name is not specified, it defaults to the name of the repo. Therefore, if you have two inputs from the same repo, you'll need to give at least one of them a unique name.

`input.atom.repo` is the `repo` to be used for the input.

`input.atom.branch` is the `branch` to watch for commits on, it may be left blank in which case `"master"` will be used.

`input.atom.commit` is the `repo` and `branch` (specified as `id`) to be used for the input, `repo` is required but `id` may be left blank in which case `"master"` will be used.

`input.atom.glob` is a glob pattern that's used to determine how the input data is partitioned. It's explained in detail in the next section.

`input.atom.lazy` controls how the data is exposed to jobs. The default is `false` which means the job will eagerly download the data it needs to process and it will be exposed as normal files on disk. If lazy is set to `true`, data will be exposed as named pipes instead and no data will be downloaded until the job opens the pipe and reads it, if the pipe is never opened then no data will be downloaded. Some applications won't work with pipes, for example if they make syscalls such as `Seek` which pipes don't support. Applications that can work with pipes should use them since they're more performant, the difference will be especially notable if the job only reads a subset of the files that are available to it. Note that `lazy` currently doesn't support datums that contain more than 10000 files.

`input.atom.from_commit` specifies the starting point of the input branch. If `from_commit` is not specified, then the entire input branch will be processed. Otherwise, only commits since the `from_commit` (not including the commit itself) will be processed.

Union Input

Union inputs take the union of other inputs. For example:

```
| inputA | inputB | inputA  inputB |
| ------ | ------ | -------------- |
| foo    | fizz   | foo            |
| bar    | buzz   | fizz           |
|        |        | bar            |
|        |        | buzz           |
```

Notice that union inputs, do not take a name and maintain the names of the sub-inputs. In the above example you would see files under `/pfs/inputA/...` and `/pfs/inputB/...`.

`input.union` is an array of inputs to union, note that these need not be `atom` inputs, they can also be `union` and `cross` inputs. Although there's no reason to take a union of unions since union is associative.

Cross Input

Cross inputs take the cross product of other inputs, in other words it creates tuples of the datums in the inputs. For example:

```
| inputA | inputB | inputA  inputB |
| ------ | ------ | -------------- |
| foo    | fizz   | (foo, fizz)    |
| bar    | buzz   | (foo, buzz)    |
|        |        | (bar, fizz)    |
|        |        | (bar, buzz)    |
```

Notice that cross inputs, do not take a name and maintain the names of the sub-inputs. In the above example you would see files under `/pfs/inputA/...` and `/pfs/inputB/...`.

`input.cross` is an array of inputs to cross, note that these need not be `atom` inputs, they can also be `union` and `cross` inputs. Although there's no reason to take a cross of crosses since cross products are associative.

OutputBranch (optional)

This is the branch where the pipeline outputs new commits. By default, it's "master".

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.

Scale-down threshold (optional)

`scale_down_threshold` specifies when the worker pods of a pipeline should be terminated.

by default, a pipeline's worker pods are always running. when `scale_down_threshold` is set, all but one worker pods are terminated after the pipeline has not seen a new job for the given duration (we still need one worker pod to subscribe to new input commits). when a new input commit comes in, the worker pods are then re-created.

`scale_down_threshold` is a string that needs to be sequence of decimal numbers with a unit suffix, such as "300ms", "1.5h" or "2h45m". valid time units are "s", "m", "h".

Incremental (optional)

Incremental, if set will cause the pipeline to be run "incrementally". This means that when a datum changes it won't be reprocessed from scratch, instead `/pfs/out` will be populated with the previous results of processing that datum and instead of seeing the full datum under `/pfs/repo` you will see only new/modified values.

Incremental processing is useful for online algorithms, a canonical example is summing a set of numbers since the new numbers can be added to the old total without having to reconsider the numbers which went into that old total. Incremental is design to work nicely with the `--split` flag to `put-file` because it will cause only the new chunks of the file to be displayed to each step of the pipeline.

Cache Size (optional)

`cache_size` controls how much cache a pipeline worker uses. In general, your pipeline's performance will increase with the cache size, but only up to a certain point depending on your workload.

The Input Glob Pattern

Each atom 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.

Multiple Inputs

It's important to note that if a pipeline takes multiple atom inputs (via cross or union) then the pipeline will not get triggered until all of the atom inputs have at least one commit on the branch.

PPS Mounts and File Access

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.

Output Formats

PFS supports data to be delimited by line, JSON, or binary blobs.

CHAPTER 28

Pachctl Command Line Tool

Pachctl is the command line interface for Pachyderm. To install Pachctl, follow the *Local Installation* instructions

Synopsis

Access the Pachyderm API.

Environment variables:

ADDRESS=<host>:<port>, the pachd server to connect to (e.g. 127.0.0.1:30650).

Options

--no-metrics	Don't report user metrics for this command
-v, --verbose	Output verbose logs

./pachctl create-pipeline

Create a new pipeline.

Synopsis

Create a new pipeline from a Pipeline Specification

```
./pachctl create-pipeline -f pipeline.json
```

Options

```
  -f, --file string       The file containing the pipeline, it can be a url or local␣
↪file. - reads from stdin. (default "-")
      --password string   Your password for the registry being pushed to.
  -p, --push-images       If true, push local docker images into the cluster registry.
  -r, --registry string   The registry to push images to. (default "docker.io")
  -u, --username string   The username to push images as, defaults to your OS␣
↪username.
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl create-repo

Create a new repo.

Synopsis

Create a new repo.

```
./pachctl create-repo repo-name
```

Options

```
  -d, --description string   A description of the repo.
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl delete-all

Delete everything.

Synopsis

Delete all repos, commits, files, pipelines and jobs. This resets the cluster to its initial state.

```
./pachctl delete-all
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl delete-branch

Delete a branch

Synopsis

Delete a branch, while leaving the commits intact

```
./pachctl delete-branch <repo-name> <branch-name>
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl delete-file

Delete a file.

Synopsis

Delete a file.

```
./pachctl delete-file repo-name commit-id path/to/file
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl delete-job

Delete a job.

Synopsis

Delete a job.

```
./pachctl delete-job job-id
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl delete-pipeline

Delete a pipeline.

Synopsis

Delete a pipeline.

```
./pachctl delete-pipeline pipeline-name
```

Options

```
      --all            delete all pipelines
      --delete-jobs    delete the jobs in this pipeline as well
      --delete-repo    delete the output repo of the pipeline as well
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl delete-repo

Delete a repo.

Synopsis

Delete a repo.

```
./pachctl delete-repo repo-name
```

Options

```
      --all     remove all repos
  -f, --force   remove the repo regardless of errors; use with care
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl deploy

Deploy a Pachyderm cluster.

Synopsis

Deploy a Pachyderm cluster.

Options

```
      --block-cache-size string       Size of pachd's in-memory cache for PFS files.␣
↪Size is specified in bytes, with allowed SI suffixes (M, K, G, Mi, Ki, Gi, etc).
      --dash-image string             Image URL for pachyderm dashboard (default
↪"pachyderm/dash:0.3.30")
      --dashboard                     Deploy the Pachyderm UI along with Pachyderm␣
↪(experimental). After deployment, run "pachctl port-forward" to connect
      --dashboard-only                Only deploy the Pachyderm UI (experimental),␣
↪without the rest of pachyderm. This is for launching the UI adjacent to an existing␣
↪Pachyderm cluster. After deployment, run "pachctl port-forward" to connect
      --dry-run                       Don't actually deploy pachyderm to Kubernetes,␣
↪instead just print the manifest.
      --dynamic-etcd-nodes int        Deploy etcd as a StatefulSet with the given␣
↪number of pods.  The persistent volumes used by these pods are provisioned␣
↪dynamically.  Note that StatefulSet is currently a beta kubernetes feature, which␣
↪might be unavailable in older versions of kubernetes.
      --etcd-cpu-request string       (rarely set) The size of etcd's CPU request,␣
↪which we give to Kubernetes. Size is in cores (with partial cores allowed and␣
↪encouraged).
      --etcd-memory-request string    (rarely set) The size of etcd's memory request.␣
↪Size is in bytes, with SI suffixes (M, K, G, Mi, Ki, Gi, etc).
      --log-level string              The level of log messages to print options are,␣
↪from least to most verbose: "error", "info", "debug". (default "info")
      --pachd-cpu-request string      (rarely set) The size of Pachd's CPU request,␣
↪which we give to Kubernetes. Size is in cores (with partial cores allowed and␣
↪encouraged).
      --pachd-memory-request string   (rarely set) The size of PachD's memory request␣
↪in addition to its block cache (set via --block-cache-size). Size is in bytes, with␣
↪SI suffixes (M, K, G, Mi, Ki, Gi, etc).
      --shards int                    (rarely set) The maximum number of pachd nodes␣
↪allowed in the cluster; increasing this number blindly can result in degraded␣
↪performance. (default 16)
      --static-etcd-volume string     Deploy etcd as a ReplicationController with one␣
↪pod.  The pod uses the given persistent volume.
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -
- ./pachctl deploy amazon - Deploy a Pachyderm cluster running on AWS.
- ./pachctl deploy custom - (in progress) Deploy a custom Pachyderm cluster configuration
- ./pachctl deploy google - Deploy a Pachyderm cluster running on GCP.

- ./pachctl deploy local - Deploy a single-node Pachyderm cluster with local metadata storage.
- ./pachctl deploy microsoft - Deploy a Pachyderm cluster running on Microsoft Azure.

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl finish-commit

Finish a started commit.

Synopsis

Finish a started commit. Commit-id must be a writeable commit.

```
./pachctl finish-commit repo-name commit-id
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl flush-commit

Wait for all commits caused by the specified commits to finish and return them.

Synopsis

Wait for all commits caused by the specified commits to finish and return them.

Examples:

```
# return commits caused by foo/XXX and bar/YYY
$ pachctl flush-commit foo/XXX bar/YYY

# return commits caused by foo/XXX leading to repos bar and baz
$ pachctl flush-commit foo/XXX -r bar -r baz
```

```
./pachctl flush-commit commit [commit ...]
```

Options

```
      --raw           disable pretty printing, print raw json
  -r, --repos value   Wait only for commits leading to a specific set of repos
↪(default [])
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl garbage-collect

Garbage collect unused data.

Synopsis

Garbage collect unused data.

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 will need to manually invoke garbage collection. The easiest way to do it is through "pachctl garbage-collecth".

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.

```
./pachctl garbage-collect
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl get-file

Return the contents of a file.

Synopsis

Return the contents of a file.

```
./pachctl get-file repo-name commit-id path/to/file
```

Options

```
  -o, --output string      The path where data will be downloaded.
  -p, --parallelism uint   The maximum number of files that can be downloaded in␣
↪parallel (default 10)
  -r, --recursive          Recursively download a directory.
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl get-logs

Return logs from a job.

Synopsis

Return logs from a job.

Examples:

```
$ pachctl get-logs --pipeline=filter

# return logs emitted by the job aedfa12aedf
$ pachctl get-logs --job=aedfa12aedf

# return logs emitted by the pipeline \"filter\" while processing /apple.txt and a␣
↪file with the hash 123aef
$ pachctl get-logs --pipeline=filter --inputs=/apple.txt,123aef
```

```
./pachctl get-logs [--pipeline=<pipeline>|--job=<job id>]
```

Options

```
      --inputs string     Filter for log lines generated while processing these files
↪(accepts PFS paths or file hashes)
      --job string        Filter for log lines from this job (accepts job ID)
      --master            Return log messages from the master process (pipeline must
↪be set).
      --pipeline string   Filter the log for lines from this pipeline (accepts
↪pipeline name)
      --raw               Return log messages verbatim from server.
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl get-object

Return the contents of an object

Synopsis

Return the contents of an object

```
./pachctl get-object hash
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl get-tag

Return the contents of a tag

Synopsis

Return the contents of a tag

```
./pachctl get-tag tag
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl glob-file

Return files that match a glob pattern in a commit.

Synopsis

Return files that match a glob pattern in a commit (that is, match a glob pattern in a repo at the state represented by a commit). Glob patterns are documented here.

Examples:

```
# Return files in repo "foo" on branch "master" that start
# with the character "A".  Note how the double quotation marks around "A*" are
# necessary because otherwise your shell might interpret the "*".
$ pachctl glob-file foo master "A*"

# Return files in repo "foo" on branch "master" under directory "data".
$ pachctl glob-file foo master "data/*"
```

```
./pachctl glob-file repo-name commit-id pattern
```

Options

```
      --raw   disable pretty printing, print raw json
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl inspect-commit

Return info about a commit.

Synopsis

Return info about a commit.

```
./pachctl inspect-commit repo-name commit-id
```

Options

```
      --raw   disable pretty printing, print raw json
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl inspect-file

Return info about a file.

Synopsis

Return info about a file.

```
./pachctl inspect-file repo-name commit-id path/to/file
```

Options

```
      --raw   disable pretty printing, print raw json
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl inspect-job

Return info about a job.

Synopsis

Return info about a job.

```
./pachctl inspect-job job-id
```

Options

```
  -b, --block   block until the job has either succeeded or failed
      --raw     disable pretty printing, print raw json
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl inspect-pipeline

Return info about a pipeline.

Synopsis

Return info about a pipeline.

```
./pachctl inspect-pipeline pipeline-name
```

Options

```
      --raw   disable pretty printing, print raw json
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl inspect-repo

Return info about a repo.

Synopsis

Return info about a repo.

```
./pachctl inspect-repo repo-name
```

Options

```
      --raw   disable pretty printing, print raw json
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl list-branch

Return all branches on a repo.

Synopsis

Return all branches on a repo.

```
./pachctl list-branch <repo-name>
```

Options

```
      --raw   disable pretty printing, print raw json
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl list-commit

Return all commits on a set of repos.

Synopsis

Return all commits on a set of repos.

Examples:

```
# return commits in repo "foo"
$ pachctl list-commit foo

# return commits in repo "foo" on branch "master"
$ pachctl list-commit foo master

# return the last 20 commits in repo "foo" on branch "master"
$ pachctl list-commit foo master -n 20

# return commits that are the ancestors of XXX
$ pachctl list-commit foo XXX

# return commits in repo "foo" since commit XXX
$ pachctl list-commit foo master --from XXX
```

```
./pachctl list-commit repo-name
```

Options

```
  -f, --from string   list all commits since this commit
  -n, --number int    list only this many commits; if set to zero, list all commits
      --raw           disable pretty printing, print raw json
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl list-file

Return the files in a directory.

Synopsis

Return the files in a directory.

```
./pachctl list-file repo-name commit-id path/to/dir
```

Options

```
      --raw   disable pretty printing, print raw json
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl list-job

Return info about jobs.

Synopsis

Return info about jobs.

Examples:

```
$ pachctl list-job

# return all jobs in pipeline foo
$ pachctl list-job -p foo

# return all jobs whose input commits include foo/XXX and bar/YYY
$ pachctl list-job foo/XXX bar/YYY

# return all jobs in pipeline foo and whose input commits include bar/YYY
$ pachctl list-job -p foo bar/YYY
```

```
./pachctl list-job [-p pipeline-name] [commits]
```

Options

```
  -p, --pipeline string   Limit to jobs made by pipeline.
      --raw               disable pretty printing, print raw json
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl list-pipeline

Return info about all pipelines.

Synopsis

Return info about all pipelines.

```
./pachctl list-pipeline
```

Options

```
      --raw   disable pretty printing, print raw json
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl list-repo

Return all repos.

Synopsis

Reutrn all repos.

```
./pachctl list-repo
```

Options

```
  -p, --provenance value   list only repos with the specified repos provenance␣
↪(default [])
      --raw                disable pretty printing, print raw json
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl login

Login to Pachyderm with your GitHub account

Synopsis

Login to Pachyderm with your GitHub account. Any resources that have been restricted to the email address registered with your GitHub account will subsequently be accessible.

```
./pachctl login
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl mount

Mount pfs locally. This command blocks.

Synopsis

Mount pfs locally. This command blocks.

```
./pachctl mount path/to/mount/point
```

Options

```
  -a, --all-commits   Show archived and cancelled commits.
  -d, --debug         Turn on debug messages.
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl port-forward

Forward a port on the local machine to pachd. This command blocks.

Synopsis

Forward a port on the local machine to pachd. This command blocks.

```
./pachctl port-forward
```

Options

```
  -k, --kubectlflags string   Any kubectl flags to proxy, e.g. --kubectlflags='--
↪kubeconfig /some/path/kubeconfig'
  -p, --port int              The local port to bind to. (default 30650)
  -x, --proxy-port int        The local port to bind to. (default 30081)
  -u, --ui-port int           The local port to bind to. (default 30080)
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl put-file

Put a file into the filesystem.

Synopsis

Put-file supports a number of ways to insert data into pfs:

```
# Put data from stdin as repo/branch/path:
$ echo "data" | pachctl put-file repo branch path

# Put data from stding as repo/branch/path and start / finish a new commit on the␣
↪branch.
$ echo "data" | pachctl put-file -c repo branch path

# Put a file from the local filesystem as repo/branch/path:
$ pachctl put-file repo branch path -f file

# Put a file from the local filesystem as repo/branch/file:
$ pachctl put-file repo branch -f file

# Put the contents of a directory as repo/branch/path/dir/file:
$ pachctl put-file -r repo branch path -f dir

# Put the contents of a directory as repo/branch/dir/file:
$ pachctl put-file -r repo branch -f dir

# Put the data from a URL as repo/branch/path:
$ pachctl put-file repo branch path -f http://host/path

# Put the data from a URL as repo/branch/path:
$ pachctl put-file repo branch -f http://host/path

# Put several files or URLs that are listed in file.
# Files and URLs should be newline delimited.
$ pachctl put-file repo branch -i file

# Put several files or URLs that are listed at URL.
# NOTE this URL can reference local files, so it could cause you to put sensitive
# files into your Pachyderm cluster.
$ pachctl put-file repo branch -i http://host/path
```

NOTE there's a small performance overhead for using a branch name as opposed to a commit ID in put-file. In most cases the performance overhead is negligible, but if you are putting a large number of small files, you might want to consider using commit IDs directly.

```
./pachctl put-file repo-name branch path/to/file/in/pfs
```

Options

```
  -c, --commit                    Put file(s) in a new commit.
  -f, --file value                The file to be put, it can be a local file or a URL.
↪ (default [-])
```

```
  -i, --input-file string         Read filepaths or URLs from a file.  If - is used,␣
↪paths are read from the standard input.
  -p, --parallelism uint          The maximum number of files that can be uploaded in␣
↪parallel. (default 10)
  -r, --recursive                 Recursively put the files in a directory.
      --split json                Split the input file into smaller files, subject to␣
↪the constraints of --target-file-datums and --target-file-bytes. Permissible values␣
↪are json and `line`.
      --target-file-bytes uint    The target upper bound of the number of bytes that␣
↪each file contains; needs to be used with --split.
      --target-file-datums uint   The upper bound of the number of datums that each␣
↪file contains, the last file will contain fewer if the datums don't divide evenly;␣
↪needs to be used with --split.
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl repo

Docs for repos.

Synopsis

Repos, short for repository, are the top level data object in Pachyderm.

```
Repos are created with create-repo.
```

```
./pachctl repo
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl run-pipeline

Run a pipeline once.

Synopsis

Run a pipeline once, optionally overriding some pipeline options by providing a pipeline spec. For example run a web scraper pipelien without any explicit input.

```
./pachctl run-pipeline pipeline-name [-f job.json]
```

Options

```
  -f, --file string   The file containing the run-pipeline spec, - reads from stdin.
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl set-branch

Set a commit and its ancestors to a branch

Synopsis

Set a commit and its ancestors to a branch.

Examples:

```
# Set commit XXX and its ancestors as branch master in repo foo.
$ pachctl set-branch foo XXX master

# Set the head of branch test as branch master in repo foo.
# After running this command, "test" and "master" both point to the
# same commit.
$ pachctl set-branch foo test master
```

```
./pachctl set-branch <repo-name> <commit-id/branch-name> <new-branch-name>
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl start-commit

Start a new commit.

Synopsis

Start a new commit with parent-commit as the parent, or start a commit on the given branch; if the branch does not exist, it will be created.

Examples:

```
# Start a new commit in repo "test" that's not on any branch
$ pachctl start-commit test

# Start a commit in repo "test" on branch "master"
$ pachctl start-commit test master

# Start a commit with "master" as the parent in repo "test", on a new branch "patch"; essentially a fork.
$ pachctl start-commit test patch -p master

# Start a commit with XXX as the parent in repo "test", not on any branch
$ pachctl start-commit test -p XXX
```

```
./pachctl start-commit repo-name [branch]
```

Options

```
  -p, --parent string   The parent of the new commit, unneeded if branch is specified and you want to use the previous head of the branch as the parent.
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl start-pipeline

Restart a stopped pipeline.

Synopsis

Restart a stopped pipeline.

```
./pachctl start-pipeline pipeline-name
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl stop-pipeline

Stop a running pipeline.

Synopsis

Stop a running pipeline.

```
./pachctl stop-pipeline pipeline-name
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl undeploy

Tear down a deployed Pachyderm cluster.

Synopsis

Tear down a deployed Pachyderm cluster.

```
./pachctl undeploy
```

Options

```
  -a, --all
Delete everything, including the persistent volumes where metadata
is stored.  If your persistent volumes were dynamically provisioned (i.e. if
you used the "--dynamic-etcd-nodes" flag), the underlying volumes will be
removed, making metadata such repos, commits, pipelines, and jobs
unrecoverable. If your persistent volume was manually provisioned (i.e. if
you used the "--static-etcd-volume" flag), the underlying volume will not be
removed.
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl unmount

Unmount pfs.

Synopsis

Unmount pfs.

```
./pachctl unmount path/to/mount/point
```

Options

```
  -a, --all   unmount all pfs mounts
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl update-pipeline

Update an existing Pachyderm pipeline.

Synopsis

Update a Pachyderm pipeline with a new Pipeline Specification

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

Options

```
  -f, --file string       The file containing the pipeline, it can be a url or local␣
↪file. - reads from stdin. (default "-")
      --password string   Your password for the registry being pushed to.
  -p, --push-images       If true, push local docker images into the cluster registry.
  -r, --registry string   The registry to push images to. (default "docker.io")
      --reprocess         If true, reprocess datums that were already processed by␣
↪previous version of the pipeline.
  -u, --username string   The username to push images as, defaults to your OS␣
↪username.
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

./pachctl version

Return version information.

Synopsis

Return version information.

```
./pachctl version
```

Options inherited from parent commands

```
      --no-metrics   Don't report user metrics for this command
  -v, --verbose      Output verbose logs
```

SEE ALSO

- ./pachctl -

Auto generated by spf13/cobra on 19-Jul-2017

CHAPTER 29

Pachyderm language clients

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
```

Python Client - `pypachy`

The Python client is a user contributed client that isn't officially maintained by the Pachyderm team. However, it implements very similar functionality to that available in the Go client or CLI.

For more info, check out `pypachy` on GitHub.

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.

Other languages

Pachyderm uses a simple protocol buffer API. Protobufs support a bunch of other languages, any of which can be used to programatically 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.