Pachyderm Documentation

Release 1.1.0

Joe Doliner

Nov 04, 2016

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 and an FAQ.

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.

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:

- *Pachyderm File System (PFS)*
- *Pachyderm Pipeline System (PPS)*
- *./pachctl*
- *FAQ*
- Use Cases

—

If you've already got a Kubernetes cluster running or would rather use AWS, GCE or Azure, check out our *Deploying on the Cloud* documentation.

—

1.1 Looking for in-depth development docs?

Learn how to *Analyze Your Data* check out more advanced Pachyderm examples such as image processing with OpenCV 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 :doc:`troubleshooting` docs for common errors, submit an issue on GitHub, join our users channel on Slack, or email us at support@pachyderm.io and we can help you right away.

2.1 Prerequisites

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

2.1.1 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.

2.1.2 Pachctl

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

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

# For Linux (64 bit):
$ curl -o /tmp/pachctl.deb -L https://pachyderm.io/pachctl.deb && sudo dpkg -i /tmp/
↪pachctl.deb
```

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

2.2 Deploy Pachyderm

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

```
pachctl deploy
```

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             DESIRED      CURRENT       AGE
etcd             1            1             6s
pachd            2            2             6s
rethink          1            1             6s
NAME             CLUSTER-IP   EXTERNAL-IP   PORT(S)                         AGE
etcd             10.0.0.45    <none>        2379/TCP,2380/TCP               6s
kubernetes       10.0.0.1     <none>        443/TCP                         6m
pachd            10.0.0.101   <nodes>       650/TCP                         6s
rethink          10.0.0.182   <nodes>       8080/TCP,28015/TCP,29015/TCP    6s
NAME             READY        STATUS        RESTARTS                        AGE
etcd-swoag       1/1          Running       0                               6s
pachd-7xyse      1/1          Running       0                               6s
pachd-gfdc6      1/1          Running       0                               6s
rethink-v5rsx    1/1          Running       0                               6s
```

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 Rethink was ready so it restarted them.

2.2.1 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             1.2.0
pachd               1.2.0
```

We're good to go!

2.3 Next Steps

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

3.1 Analyzing Log Lines from a Fruit Stand

In this guide you're going to create a Pachyderm pipeline to process transaction logs from a fruit stand. We'll use two standard unix tools, `grep` and `awk` to do our processing. 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 materialize 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.

3.1.1 Prerequisites

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

3.1.2 Create a Repo

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

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

```
$ pachctl create-repo data

# See the repo we just created
$ pachctl list-repo
data
```

3.1.3 Adding Data to Pachyderm

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

Let's start by just adding a file to a new commit. We've provided a sample data file for you to use in our GitHub repo – it's a list of purchases from a fruit stand.

We'll use the `put-file` command along with two flags, `-c` and `-f`. `-f` can take either a local file or a URL, in our case, the sample data on GitHub.

We also specificy the repo name "data" and the branch name "master".

```
$ pachctl put-file data master sales -c -f https://raw.githubusercontent.com/
↪pachyderm/pachyderm/v1.2.1/doc/examples/fruit_stand/set1.txt
```

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 we used above specifies that we want to start a new commit, add data, and finish the commit in a convenient one-liner.

Finally, we can see the data we just added to Pachyderm.

```
# If we list the repos, we can see that there is now data
$ pachctl list-repo
NAME                CREATED             SIZE
data                12 minutes ago       874 B

# We can view the commit we just created
pachctl list-commit data
BRANCH              REPO/ID         PARENT              STARTED             FINISHED ␣
↪           SIZE
master              data/master/0   <none>              6 minutes ago       6 minutes␣
↪ago       874 B

# We can also view the contents of the file that we just added
$ pachctl get-file data master sales
orange          4
banana          2
banana          9
orange          9
...
```

3.1.4 Create a Pipeline

Now that we've got some data in our repo, it's time to do something with it. `Pipelines` are the core primitive for Pachyderm's processing system (pps) and they're specified with a JSON encoding. For this example, we've already created the pipeline for you and it can be found at examples/fruit_stand/pipeline.json on Github. Please open a new tab to view the pipeline while we talk through it.

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 just using simple shell commands as we're doing here.

For now, we're going to create a pipeline with 2 transformations in it. The first transformation filters the sales logs into separate records for apples, oranges and bananas. The second step sums these sales numbers into a final sales count.

```
+----------+     +--------------+     +------------+
|input data| --> |filter pipline| --> |sum pipeline|
+----------+     +--------------+     +------------+
```

In the first step of this pipeline, we are grepping for the terms "apple", "orange", and "banana" and writing that line to the corresponding file. Notice we read data from `/pfs/data` (`/pfs/[input_repo_name]`) and write data to `/pfs/out/`. These are special local directories that Pachyderm creates within the container for you. All the input data will be found in `/pfs/[input_repo_name]` and your code should always write to `/pfs/out`.

The second step of this pipeline takes each file, removes the fruit name, and sums up the purchases. The output of our complete pipeline is three files, one for each type of fruit with a single number showing the total quantity sold.

Now let's create the pipeline in Pachyderm:

```
$ pachctl create-pipeline -f https://raw.githubusercontent.com/pachyderm/pachyderm/v1.
↪2.1/doc/examples/fruit_stand/pipeline.json
```

3.1.5 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.

You can view the job with:

```
$ pachctl list-job
       ID                                OUTPUT                                                                 ↩
↪ STARTED             DURATION            STATE
       90c74896fd227f319c3c19459aa7a22b  sum/e4060e15948c4b7b89947a02eace5dca/0                                 ↩
↪ 2 minutes ago       Less than a second  success
       67c30d70ba9d2179aa133255f5dc81db  filter/d737e9b7cfae40d4aa8a8871cdb9f783/0                               ↩
↪ 3 minutes ago       2 seconds           success
```

Every pipeline creates a corresponding repo with the same name where it stores its output results. In our example, the "filter" transformation created a repo called "filter" which was the input to the "sum" transformation. The "sum" repo contains the final output files.

```
$ pachctl list-repo
NAME                CREATED             SIZE
sum                 2 minutes ago       12 B
filter              2 minutes ago       200 B
data                19 minutes ago      874 B
```

3.1.6 Reading the Output

We can read the output data from the "sum" repo in the same fashion that we read the input data (except now we need to use an explicit commitID because the "sum" repo doesn't have a "master" branch:

```
$ pachctl get-file sum e4060e15948c4b7b89947a02eace5dca/0 apple
133
```

3.1.7 Processing More Data

Pipelines will also automatically process the data from new commits as they are created. Think of pipelines as being subscribed to any new commits that are finished on their input repo(s). Also similar to Git, commits have a parental structure that track how files change over time. In this case we're going to be adding more data to the same file "sales."

In our fruit stand example, this could be making a commit every hour with all the new purchases that happened in that timeframe.

Let's create a new commit with our previous commit as the parent and add more sample data (set2.txt) to "sales":

```
$ pachctl put-file data master sales -c -f https://raw.githubusercontent.com/
↪pachyderm/pachyderm/v1.2.1/doc/examples/fruit_stand/set2.txt
```

Adding a new commit of data will automatically trigger the pipeline to run on the new data we've added. We'll see a corresponding commit to the output "sum" repo with files "apple", "orange" and "banana" each containing the cumulative total of purchases. Let's read the "apples" file again and see the new total number of apples sold.

```
$ pachctl get-file sum 4092f4675650476ab0a3fde5b7780316/0 apple
324
```

One thing that's interesting to note is that our pipeline is completely incremental. Since `grep` is a `map` operation, Pachyderm will only `grep` the new data from set2.txt instead of re-filtering all the data. If you look back at the "sum" pipeline, you'll notice the `method` and that our code uses `/prev` to compute the sum incrementally based upon our previous commit. You can learn more about incrementally in our advanced *Incrementality* docs.

We can view the parental structure of the commits we just created.

```
$ pachctl list-commit data
BRANCH              REPO/ID             PARENT              STARTED
↪FINISHED           SIZE
master              data/master/0       <none>              19 minutes ago
↪minutes ago        874 B
master              data/master/1       master/0            2 minutes ago
↪minutes ago        874 B
```

3.1.8 Exploring the File System

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.

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 &
```

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.

```
# We can see our repos
$ ls ~/pfs
```

```
data   filter  sum

# And commits
$ ls ~/pfs/sum
4092f4675650476ab0a3fde5b7780316/1     4092f4675650476ab0a3fde5b7780316/0
```

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

3.1.9 Next Steps

You've now got Pachyderm running locally with data and a pipeline! If you want to keep playing with Pachyderm locally, here are some ideas to expand on your working setup.

- Write a script to stream more data into Pachyderm. We already have one in Golang for you on GitHub if you want to use it.
- Add a new pipeline that does something interesting with the "sum" repo as an input.
- Add your own data set and `grep` for different terms. This example can be generalized to generic word count.

You can also start learning some of the more advanced topics to develop analysis in Pachyderm:

- *Deploying on the Cloud*
- *Getting Your Data into Pachyderm* from other sources
- *Creating Analysis Pipelines* using your own code

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 dev@pachyderm.io if you want to show off anything nifty you've created!

CHAPTER 4

Troubleshooting

Below a list of common errors we run accross with users trying to run Pachyderm locally and following the *Beginner Tutorial*.

One of the first things you should do is check the logs:

```
$ pachctl list-job
$ get-logs <jobID>
```

If the problem is with deployment of Pachyderm, Kubernetes has their own logs too:

```
$ kubectl get all
$ kubectl logs pod_name
```

4.1 Profiling

Sometimes if your pachyderm cluster is misbehaving profiling it can give more information about what's going on. Pachyderm exposes the standard go profiling tool pprof over http. You can access it at `http://host:30651` host should be the same host you point `pachctl` at. To learn more about using pprof, check **'here<https://golang.org/pkg/net/http/pprof/>'_**

4.2 Common Errors

4.2.1 Error: error messages with "cannot unmarshal"

Solution: This is usually due to a version mismatch. Start with `pachctl version` and make sure your client and server version are matching.

If you got the error when running a pipeline, it's likely you're using the wrong version of the pipeline spec. For example, `json: cannot unmarshal bool into Go value of type pps.Incremental` is because the pipeline spec between v1.1 and v1.2 changed the type for incrememental from `bool` to `string`. Refer to *Pipeline Specification* to check that yours is correct.

4.2.2 Error: Job status is stuck in "pulling" state

Solution: This means that Pachyderm can't find the image specified in your pipeline. It's possible you have a typo in your pipeline spec. More likely, the image isn't available on DockerHub, your DOcker registry, or locally to the

Pachyderm pods. If you're running Pachyderm in Minikube, you need ot make sure any images you've built locally are accessible to Pachyderm within the VM.

4.2.3 Error: `pipeline <NAME> already exists`

or `repo <NAME> already exists`

Solution: Pipelines and repos need to have unique names. The errors happens most commonly when your first attempt at creating or running a pipeline failed and you try to do `create-pipeline` again. There are a bunch of solutions.

1. Delete the previous pipelines and associated repos. `pachctl delete-repo <name>` and `pachctl delete-pipeline <name>`.
2. Change the "name" field in your pipeline spec JSON file.
3. Use the `update-pipeline` command. `update-pipeline` is a bit more complicated to use so make sure to read the *Updating Pipelines* docs.

4.2.4 Error: A pipeline using `grep` fails with a [1] error code.

Solution: Little known fact, `grep` errors with [1] if it does not find any results. For our fruit stand example, if we did grepped for grapes, the pipeline would end up failing. It's really easy to solve this by adding "acceptReturnCode": [1] in stdin of your pipeline spec.

Technically, this [1] error could be coming from some other part of your code, but the most common culprit we've seen if you're running simple examples is grep.

4.2.5 Error: SOMETHING IS F***ED AND I WANT TO START OVER!

Solution: We got your back!

```
# Delete all pipelines and repos in Pachyderm
$ pachctl delete-all

# Archive all commits in all repos. This is ideal if you have a bunch of garbage␣
↪data, but want
# to keep your pipelines and repos intact. Your old data is still available using␣
↪list-commit -a.
$ pachctl archive-all

# Remove Pachyderm from your kubernetes cluster
$ pachctl deploy --dry-run | kubectl delete -f -

# Kill the entire minikube VM and restart. Don't skip the minikube delete step
# because it keeps around some weird intermediate state.
$ minikube stop
$ minikube delete
$ minikube start
```

CHAPTER 5

Analyze Your Data

This section of documentation covers everything you'll need to know to deploy a working Pachyderm cluster and build your own analysis to process whatever data you want.

If you're brand new to Pachyderm, you should check out our *Getting Started* documentation to install Pachyderm locally and learn the basic concepts.

Deploying on the Cloud: Get Pachyderm deployed on AWS, GCE, Azure, or OpenShift.

Getting Your Data into Pachyderm: Get your own data into Pachyderm.

Creating Analysis Pipelines: Get your code running in Pachyderm and processing data.

Pipeline Specification: A complete reference on the advanced features of Pachyderm pipelines.

Getting your Results: Read out results from specific input commits.

Updating Pipelines: Interate on pipelines as you learn from your data.

5.1 Usage Metrics

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

CHAPTER 6

Deploying on the Cloud

6.1 Intro

Pachyderm is built on Kubernetes. As such, Pachyderm can run on any platform that supports Kubernetes. This guide covers the following commonly used platforms:

- *Google Cloud Platform*
- *AWS*
- *Microsoft Azure*
- *OpenShift*

6.2 Google Cloud Platform

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

6.2.1 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. 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` .

If you do not already have `kubectl` installed, after the SDK is installed, run:

```
$ gcloud components install kubectl
```

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

6.2.2 Deploy Kubernetes

To create a new Kubernetes cluster in GKE, just 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}

# 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
```

This may take a few minutes to start up. You can check the status on the GCP Console.

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

Check to see that your cluster is up and running:

```
$ kubectl get all
NAME         CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   10.3.240.1   <none>        443/TCP   10m
```

6.2.3 Deploy Pachyderm

Set up the Storage Infrastructure

Pachyderm needs a GCS bucket and a persistent disk to function correctly.

Here are the parameters to create these resources:

```
# 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]

# 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
↪"]
```

And then run:

```
$ gsutil mb gs://${BUCKET_NAME}
$ gcloud compute disks create --size=${STORAGE_SIZE}GB ${STORAGE_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 used for interacting with a Pachyderm cluster.

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

# For Linux (64 bit):
$ curl -o /tmp/pachctl.deb -L https://pachyderm.io/pachctl.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             1.2.2
pachd               (version unknown) : error connecting to pachd server at address␣
↪(0.0.0.0:30650): context deadline exceeded.
```

Start Pachyderm

Now we're ready to boot up Pachyderm:

```
$ pachctl deploy google ${BUCKET_NAME} ${STORAGE_NAME} ${STORAGE_SIZE}
```

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                    DESIRED         CURRENT           AGE
etcd                    1               1                 1m
pachd                   2               2                 1m
rethink                 1               1                 1m
NAME                    CLUSTER-IP      EXTERNAL-IP       PORT(S)                                                  ␣
↪AGE
etcd                    10.3.253.161    <none>            2379/TCP,2380/TCP                                        ␣
↪1m
kubernetes              10.3.240.1      <none>            443/TCP                                                  ␣
↪47m
pachd                   10.3.254.31     <nodes>           650/TCP,651/TCP                                          ␣
↪1m
rethink                 10.3.241.56     <nodes>           8080/TCP,28015/TCP,29015/TCP                             ␣
↪1m
NAME                    READY           STATUS            RESTARTS                                                 ␣
↪AGE
etcd-1mv3v              1/1             Running           0                                                        ␣
↪1m
pachd-6vjpc             1/1             Running           3                                                        ␣
↪1m
pachd-nxj54             1/1             Running           3                                                        ␣
↪1m
rethink-e4v60           1/1             Running           0                                                        ␣
↪1m
NAME                    STATUS          VOLUME            CAPACITY                                                 ␣
↪ACCESSMODES   AGE
rethink-volume-claim    Bound           rethink-volume    10Gi                                                     ␣
↪RWO            1m
```

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 Rethink 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 portforward &
```

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             1.2.0
pachd               1.2.0
```

6.3 Amazon Web Services (AWS)

6.3.1 Prerequisites

- Make sure you have the AWS CLI installed and have your AWS credentials configured.

6.3.2 Deploy Kubernetes

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

NOTE: If you've already got a Kubernetes cluster running, you may see the error `An error occurred (InvalidIPAddress.InUse) when calling the RunInstances operation: Address 172.20.0.9 is in use`. You can terminate the old cluster with `kubernetes/cluster/kube-down.sh` and then rerun the script.

NOTE: If you already had kubectl set up from the minikube demo, kubectl will now be talking to your aws cluster. You can switch back to talking to minikube with:

```
kubectl config use-context minikube
```

Now we've got Kubernetes up and running, it's time to deploy Pachyderm!

6.3.3 Deploy Pachyderm

Before we deploy Pachyderm, we need to add some storage resources to our cluster so that Pachyderm has a place to put data.

Set up the Storage Infrastructure

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

Here are the parameters to set up 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"]
```

And then 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"). You can also view it in the aws console or with `aws ec2 describe-volumes`. Export the volume-id:

```
$ STORAGE_NAME=[volume id]
```

Now 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'
```

Install Pachctl

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

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

# For Linux (64 bit):
$ curl -o /tmp/pachctl.deb -L https://pachyderm.io/pachctl.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             1.2.0
pachd               (version unknown) : error connecting to pachd server at address␣
↪(0.0.0.0:30650): context deadline exceeded.
```

```
#### Start Pachyderm

First get a set of [temporary AWS credentials](http://docs.aws.amazon.com/IAM/latest/
↪UserGuide/id_credentials_temp.html) by using this command:

```shell
$ aws sts get-session-token
```

Then set these variables:

```
$ AWS_ID=[access key ID]

$ AWS_KEY=[secret access key]

$ AWS_TOKEN=[session token]
```

Run the following command to deploy your Pachyderm cluster:

```
$ pachctl deploy amazon ${BUCKET_NAME} ${AWS_ID} ${AWS_KEY} ${AWS_TOKEN} ${AWS_REGION}
↪ ${STORAGE_NAME} ${STORAGE_SIZE}
```

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                    DESIRED         CURRENT           AGE
etcd                    1               1                 17m
pachd                   2               2                 17m
rethink                 1               1                 17m
NAME                    CLUSTER-IP      EXTERNAL-IP       PORT(S)
↪AGE
etcd                    10.0.255.155    <none>            2379/TCP,2380/TCP
↪17m
kubernetes              10.0.0.1        <none>            443/TCP
↪4h
pachd                   10.0.43.148     <nodes>           650/TCP,651/TCP
↪17m
rethink                 10.0.249.8      <nodes>           8080/TCP,28015/TCP,29015/TCP
↪17m
NAME                    READY           STATUS            RESTARTS
↪AGE
etcd-04jbq              1/1             Running           0
↪17m
pachd-7a8sp             1/1             Running           2
↪17m
pachd-9egd7             1/1             Running           3
↪17m
rethink-xd7sc           1/1             Running           0
↪17m
NAME                    STATUS          VOLUME            CAPACITY
↪ACCESSMODES   AGE
rethink-volume-claim    Bound           rethink-volume    10Gi
↪RWO            17m
```

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 Rethink 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             1.2.3
pachd               1.2.3
```

6.4 Microsoft Azure

6.4.1 Prerequisites

- Install Azure CLI >= 0.10.6
- Install jq

6.4.2 Deploy Kubernetes

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

6.4.3 Deploy Pachyderm

Set up the Storage Infrastructure

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
$ AZURE_RESOURCE_GROUP=[The name of the resource group will all the Azure resources␣
↪will be stored]

$ AZURE_LOCATION=[The Azure region of your Kubernetes cluster. e.g. "West US2"]

# Needs to be globally unique across the entire Azure location
$ AZURE_STORAGE_NAME=[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
```

And then run:

```
$ azure group create --name ${AZURE_RESOURCE_GROUP} --location ${AZURE_LOCATION
$ azure storage account create ${AZURE_STORAGE_NAME} --location ${AZURE_LOCATION} --
↪resource-group ${AZURE_RESOURCE_GROUP} --sku-name LRS --kind Storage
```

```
# Retrieve the Azure Storage Account Key
$ AZURE_STORAGE_KEY=`azure storage account keys list ${AZURE_STORAGE_NAME} --resource-
↪group ${AZURE_RESOURCE_GROUP} --json | jq .[0].value`

# Build the microsoft_vhd container.
$ make docker-build-microsoft-vhd

# Create an empty data disk in the "disks" container
$ docker run -it microsoft_vhd ${AZURE_STORAGE_NAME} ${AZURE_STORAGE_KEY} "disks" $
↪{STORAGE_NAME}
```

Record the *volume-uri* that is printed and export it as:

```
$ STORAGE_VOLUME_URI=[volume uri / output of docker run command]
```

To check that everything has been setup correctly, try:

```
$ azure storage account list
# should see a number of storage accounts, including the one specified with ${AZURE_
↪STORAGE_NAME}

$ azure storage blob list --account-name ${AZURE_STORAGE_NAME} --account-key ${_AZURE_
↪STORAGE_KEY}
# should see a disk with the name ${STORAGE_NAME}
```

Install Pachctl

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

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

# For Linux (64 bit):
$ curl -o /tmp/pachctl.deb -L https://pachyderm.io/pachctl.deb && 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             1.2.2
pachd               (version unknown) : error connecting to pachd server at address␣
↪(0.0.0.0:30650): context deadline exceeded.
```

Start Pachyderm

Now we're ready to boot up Pachyderm:

```
$ pachctl deploy microsoft ${CONTAINER_NAME} ${AZURE_STORAGE_NAME} ${AZURE_STORAGE_
↪KEY} ${STORAGE_VOLUME_UR} ${STORAGE_SIZE}
```

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                    DESIRED         CURRENT           AGE
etcd                    1               1                 1m
pachd                   2               2                 1m
rethink                 1               1                 1m
NAME                    CLUSTER-IP      EXTERNAL-IP       PORT(S)
 ↪AGE
etcd                    10.3.253.161    <none>            2379/TCP,2380/TCP
 ↪1m
kubernetes              10.3.240.1      <none>            443/TCP
 ↪47m
pachd                   10.3.254.31     <nodes>           650/TCP,651/TCP
 ↪1m
rethink                 10.3.241.56     <nodes>           8080/TCP,28015/TCP,29015/TCP
 ↪1m
NAME                    READY           STATUS            RESTARTS
 ↪AGE
etcd-1mv3v              1/1             Running           0
 ↪1m
pachd-6vjpc             1/1             Running           3
 ↪1m
pachd-nxj54             1/1             Running           3
 ↪1m
rethink-e4v60           1/1             Running           0
 ↪1m
NAME                    STATUS          VOLUME            CAPACITY
 ↪ACCESSMODES   AGE
rethink-volume-claim    Bound           rethink-volume    10Gi
 ↪RWO            1m
```

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 Rethink 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 portforward &
```

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             1.2.0
pachd               1.2.0
```

6.5 OpenShift

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

1. Make sure that privilege containers are allowed (they are not allowed by default): `oc edit scc` and set `allowPrivilegedContainer: true` everywhere.
2. Remove `hostPath` everywhere from your cluster manifest (e.g. `etc/kube/pachyderm-versioned.json` if you are deploying locally).

Problems related to OpenShift deployment are tracked in this issue: https://github.com/pachyderm/pachyderm/issues/336

6.6 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 7

Getting Your Data into Pachyderm

If you're running Pachyderm in the cloud, data in Pachyderm is backed an object store such as S3 or GCS. Files in Pachyderm are content-addressed as part of how we buid our version control semantics and are therefore not "human-readable." We recommend you give Pachyderm its own bucket.

There a bunch of different ways to get your data into Pachyderm.

PFS Mount: This is a "toy" method for getting data into Pachyderm if you just have some local files (or dummy files) and you just want to test things out.

Pachctl CLI: This is the best option for real use cases and scripting the input process.

Golang Client: Ideal for Golang users who want to script the file input process.

Other Language Clients: Pachyderm uses a protobuf API which supports many other languages, we just haven't built full clients yet.

7.1 PFS Mount

This is the easiest method if you just have some local files (or dummy files) and you just want to test things out in Pachyderm. This is NOT a production method for getting data into Pachyderm.

Pachyderm allows you to mount data in the distributed file system locally using and explore it using FUSE.

FUSE comes pre-installed on most Linux distributions. For OS X, you'll need to install [OSX FUSE](https://osxfuse.github.io/)

First create the mount point:

```
$ mkdir ~/pfs
```

And then mount it:

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

This will mount pfs on `~/pfs` you can inspect the filesystem like you would any other local filesystem using `ls` or a web browser.

Once you have pfs mounted, you can add files to Pachyderm via whatever method you prefer to manipulate a local file system: `mv` , `cp` , `>` , `|` , etc.

Don't forget, you'll need create a repo and commit in Pachyderm first with:

```
# Create a repo called "data"
$ pachctl create-repo data

# Start a commit on repo "data"
$ pachctl start-commit data



Now add whatever files you want to ``~/pfs/<repo_name>/<commit_ID>/<file_name>``.
```

7.2 Pachctl CLI

The pachctl CLI is the primary method of interaction with Pachyderm. To get data into Pachyderm, you should use the `put-file` command. Below are a example uses of `put-file` . Go to ../pachctl/pachctl_put-file for complete documentation.

Note: 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 the commit in addition to putting data as a one-line command.

Add a single file:

```
$ pachctl put-file <repo> <branch> -f <file>
```

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

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

Put data from a URL:

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

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

```
$ pachctl put-file <repo> <branch> -i <file>
```

Pipe data from stdin into a file:

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

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

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

7.3 Golang Client

For any Go users, we've built a Golang client so you can easily script Pachyderm commands. Check out the autogenerated godocs on `put-file` .

7.4 Other Language Clients

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.

CHAPTER 8

Creating Analysis Pipelines

There are three steps to writing analysis in Pachyderm.

1. Write your code
2. Generate a Docker image with your code
3. Create/add your pipeline in Pachyderm

8.1 Writing your analysis code

Analysis code in Pachyderm can be written using any languages or libraries you want. At the end of the day, all the dependencies and code will be built into a container so it can run anywhere. 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 data from the local file system.

For reading, Pachyderm will automatically mount each input data repo as `/pfs/<repo_name>`. Your analysis code doesn't have to deal with distributed processing as Pachyderm will automatically shard the input data across parallel containers. For example, if you've got four containers running your Python code, `/pfs/<repo_name>` in each container will only have 1/4 of the data. You 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.

All writes simply need to go to `/pfs/out`. This is a special directory that is available in every container. As with reading, your code doesn't have to manage parallelization, just write data to `/pfs/out` and Pachyderm will make sure it all ends up in the correct place.

8.2 Building a Docker Image

As part of a pipeline, you need to specify a Docker image including the code you want to run.

There are two ways to construct the image. Both require some familiarity with Dockerfiles.

In short, you need to include a bit of Pachyderm code, called the Job Shim, in your image, but a Dockerfile can only have a single `FROM` directive. Therefore, you can either add your code to our job-shim image or you can add our job-shim code to your own image.

8.2.1 Using the Pachyderm Job Shim

Use this method if your dependencies are simple. Just base your image off of Pachyderm's job shim image:

```
FROM pachyderm/job-shim:latest
```

Here is an example where the transformation code was written in Go, so in addition to using the job-shim image, this Dockerfile installed go1.6.2 and compiled the program needed for the transformation.

8.2.2 Adding the Job-shim Code to Your Image

Use this method if your dependencies are pretty complex or you're using a published 3rd-party image such as TensorFlow.

In this case, the `FROM` directive will specify the 3rd party image of your choice and then you'll add the Pachyderm code to your Dockerfile. Below is the code you need to add (you can also view it on GitHub).

```
FROM `Your Image`

# then ...
# Install FUSE
RUN \
  apt-get update -yq && \
  apt-get install -yq --no-install-recommends \
    git \
    ca-certificates \
    curl \
    fuse && \
  apt-get clean && \
  rm -rf /var/lib/apt

# Install Go 1.6.0 (if you don't already have it in your base image)
RUN \
  curl -sSL https://storage.googleapis.com/golang/go1.6.linux-amd64.tar.gz | tar -C /
↪usr/local -xz && \
  mkdir -p /go/bin
ENV PATH /usr/local/go/bin:/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin:/bin:/
↪sbin
ENV GOPATH /go
ENV GOROOT /usr/local/go

# Install Pachyderm job-shim
RUN go get github.com/pachyderm/pachyderm && \
    go get github.com/pachyderm/pachyderm/src/server/cmd/job-shim && \
    cp $GOPATH/bin/job-shim /job-shim
```

8.2.3 Distributing your Image

Unless Pachyderm is running on the same Docker host that you used to build your image you'll need to use a registry to get your image into the cluster. `pachctl` can help you with this by way of the `--push-images` flag which is accepted by `create-pipeline` and `update-pipeline`. By default the flag will attempt to push the image to a registry that's started inside of the Kubernetes cluster when you launch Pachyderm. You can use it with other registries such as DockerHub and Google Container Registry. Learn more about `--push-images` from the `create-pipeline` docs.

8.3 Creating a Pipeline

Now that you've got your code and image built, the final step is to add a pipeline manifest to Pachyderm. Pachdyerm pipelines are described using a JSON file. There are four main components to a pipeline: name, transform, parallelism and inputs. Detailed explanations of parameters and how they work can be found in the pipeline_spec.

Here's a template pipeline:

```
{
  "pipeline": {
    "name": "my-pipeline"
  },
  "transform": {
    "image": "my-image",
    "cmd": [ "my-binary", "arg1", "arg2"],
    "stdin": [
        "my-std-input"
    ]
  },
  "parallelism": "4",
  "inputs": [
    {
      "repo": {
        "name": "my-input"
      },
      "method": "map"
    }
  ]
}
```

After you create the JSON manifest, you can add the pipeline to Pachyderm using:

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

`-f` can also take a URL if your JSON manifest is hosted on GitHub for instance. Keeping pipeline manifests under version control too 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 your code on *every* finished commit in the input repo(s) as well as *all future commits* that happen after the pipeline is created.

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.

CHAPTER 9

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.

9.1 JSON Manifest Format

```
{
  "pipeline": {
    "name": string
  },
  "transform": {
    "image": string,
    "cmd": [ string ],
    "stdin": [ string ]
    "env": {
        "foo": "bar"
    },
    "secrets": [ {
        "name": "secret_name",
        "mountPath": "/path/in/container"
    } ]
  },
  "parallelism_spec": {
    "strategy": "CONSTANT"|"COEFFICIENT"
    "constant": int         // if strategy == CONSTANT
    "coefficient": double   // if strategy == COEFFICIENT
  }
  "inputs": [
    {
      "repo": {
        "name": string
      },
      "runEmpty": false,
      "method": "map"/"reduce"/"global"
      // alternatively, method can be specified as an object.
      // this is only for advanced use cases; most of the time, one of the three
      // strategies above should suffice.
      "method": {
        "partition": "BLOCK"/"FILE"/"REPO",
        "incremental": "NONE"/"DIFF"/"FILE",
      }
```

```
    }
  ]
}
```

9.1.1 Name

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

9.1.2 Transform

`transform.image` is the name of the Docker image that your jobs run in. Currently, this image needs to inherit from a Pachyderm-provided image known as `job-shim`.

`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.

9.1.3 Parallelism Spec

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

If you use the `CONSTANT` strategy, Pachyderm will start a number of workers that you give it. To use this strategy, set the field `strategy` to `CONSTANT`, and set the field `constant` to an integer value (e.g. `10` to start 10 workers).

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

Note: Pachyderm treats this config as an upper bound. Pachyderm may choose to start fewer workers than specified if the pipeline's input data set is small or otherwise doesn't parallelize well (for example, if you use an input method of file and the input repo only has one file in it).

9.1.4 Inputs

`inputs` specifies a set of 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.

`inputs.runEmpty` specifies what happens when an empty commit (i.e. no data) comes into the input repo of this pipeline (for example, if an input pipeline produced no data). If this flag is set to true, Pachyderm will still run your pipeline even if it has no new input data to process. Specifically: if this flag is set to false (the default), then an empty commit won't trigger a job; if set to true, an empty commit will trigger a job.

`inputs.method` specifies two different properties:

- Partition unit: How input data will be partitioned across parallel containers.
- Incrementality: Whether the entire all of the data or just the new data (diff) is processed.

The next section explains input methods in detail.

9.1.5 Pipeline Input Methods

For each pipeline input, you may specify a "method". A method dictates exactly what happens in the pipeline when a commit comes into the input repo.

A method consists of two properties: partition unit and incrementality.

Partition Unit

Partition unit ("BLOCK", "FILE", or "REPO") specifies the granularity at which input data is parallelized across containers. It can be of three values:

1. `BLOCK` : different blocks of the same file may be parelleized across containers.
2. `FILE` : the files and/or directories residing under the root directory (/) must be grouped together. For instance, if you have four files in a directory structure like:

```
/foo
/bar
/buzz
   /a
   /b
```

then there are only three top-level objects, `/foo` , `/bar` , and `/buzz` , each of which will remain grouped in the same container.

1. `REPO` : the entire repo. In this case, the input won't be partitioned at all.

Incrementality

Incrementality ("NONE", "DIFF" or "FILE") describes what data needs to be available when a new commit is made on an input repo. Namely, do you want to process *only the new data* in that commmit (the "DIFF"), only files with any new data ("FILE"), or does all of the data need to be reprocessed ("NONE")?

For instance, if you have a repo with the file `/foo` in commit 1 and file `/bar` in commit 2, then:

- If the input incrementality is "DIFF", the first job sees file `/foo` and the second job sees file `/bar` .
- If the input is non-incremental("NONE"), every job sees all the data. The first job sees file `/foo` and the second job sees file `/foo` and file `/bar` .
- "File" (Top-level objects) means that if any part in a file (or alternatively any file within a directory) changes, then show all the data in that file (directory). For example, you may have vendor data files in separate directories by state – the California directory contains a file for each california vendor, etc. `Incremental: "file"` would mean that your job will see the entire directory if at least one file in that directory has changed. If only one vendor file in the whole repo was was changed and it was in the Colorado directory, all Colorado vendor files would be present, but that's it.

Combining Partition unit and Incrementality

For convenience, we have defined aliases for the three most commonly used (and most familiar) input methods: "map", "reduce", and "global".

- A "map" (BLOCK + DIFF), for example, can partition files at the block level and jobs only need to see the new data.
- "Reduce" (FILE + NONE) as it's typically seen in Hadoop, requires all parts of a file to be seen by the same container ("FILE") and your job needs to reprocess *all* the data in the whole repo ("NONE").
- "Global" (REPO + NONE), means that the entire repo needs to be seen by *every* container. This is commonly used if you had a repo with just parameters, and every container needed to see all the parameter data and pull out the ones that are relevant to it.

They are defined below:

	Partition Unit		
Incrementality	"Block"	"FILE" (Top-lvl Obj)	"REPO"
"NONE" (non-incremental)		"reduce"	"global"
"DIFF" (incremental)	"map"		
"FILE" (top-lvl object)			

Defaults

If no method is specified, the `map` method (BLOCK + DIFF) is used by default.

9.1.6 Multiple Inputs

A pipeline is allowed to have multiple inputs. The important thing to understand is what happens when a new commit comes into one of the input repos. In short, a pipeline processes the **cross product** of its inputs. We will use an example to illustrate.

Consider a pipeline that has two input repos: `foo` and `bar`. `foo` uses the `file/incremental` method and `bar` uses the `reduce` method. Now let's say that the following events occur:

```
1. PUT /file-1 in commit1 in foo -- no jobs triggered
2. PUT /file-a in commit1 in bar -- triggers job1
3. PUT /file-2 in commit2 in foo -- triggers job2
4. PUT /file-b in commit2 in bar -- triggers job3
```

The first time the pipeline is triggered will be when the second event completes. This is because we need data in both repos before we can run the pipeline.

Here is a breakdown of the files that each job sees:

```
job1:
    /pfs/foo/file-1
    /pfs/bar/file-a

job2:
```

```
    /pfs/foo/file-2
    /pfs/bar/file-a

job3:
    /pfs/foo/file-1
    /pfs/foo/file-2
    /pfs/bar/file-a
    /pfs/bar/file-b
```

`job1` sees `/pfs/foo/file-1` and `/pfs/bar/file-a` because those are the only files available.

`job2` sees `/pfs/foo/file-2` and `/pfs/bar/file-a` because it's triggered by commit2 in `foo`, and `foo` uses an incremental input method (`file/incremental`).

`job3` sees all the files because it's triggered by commit2 in `bar`, and `bar` uses a non-incremental input method (`reduce`).

9.2 Examples

```
{
  "pipeline": {
    "name": "my-pipeline"
  },
  "transform": {
    "image": "my-image",
    "cmd": [ "my-binary", "arg1", "arg2"],
    "stdin": [
        "my-std-input"
    ]
  },
  "parallelism": "4",
  "inputs": [
    {
      "repo": {
        "name": "my-input"
      },
      "method": "map"
    }
  ]
}
```

This pipeline runs when the repo `my-input` gets a new commit. The pipeline will spawn 4 parallel jobs, each of which runs the command `my-binary` in the Docker image `my-image`, with `arg1` and `arg2` as arguments to the command and `my-std-input` as the standard input. Each job will get a set of blocks from the new commit as its input because `method` is set to `map`.

9.3 PPS Mounts and File Access

9.3.1 Mount Paths

The root mount point is at `/pfs`, which contains:

- `/pfs/input_repo` which is where you would find the latest commit from each input repo you specified.

  - Each input repo will be found here by name
  - Note: Unlike when mounting locally for debugging, there is no `Commit` ID in the path. This is because the commit will always change, and the ID isn't relevant to the processing. The commit that is exposed is configured based on the `incrementality` flag above
- `/pfs/out` which is where you write any output
- `/pfs/prev` which is this `Job` or `Pipeline` 's previous output, if it exists. (You can think of it as this job's output commit's parent).

9.3.2 Output Formats

PFS supports data to be delimited by line, JSON, or binary blobs. Refer here for more information on delimiters

9.4 Environment Variables

When the pipeline runs, the input and output commit IDs are exposed via environment variables:

- `$PACH_OUTPUT_COMMIT_ID` contains the output commit of the job itself
- For each of the job's input repositories, there will be a corresponding environment variable w the input commid ID:
  - e.g. if there are two input repos `foo` and 'bar, the following will be populated:
    * `$PACH_FOO_COMMIT_ID`
    * `$PACH_BAR_COMMIT_ID`

9.5 Flash-crowd behavior

In distributed systems, a flash-crowd behavior occurs when a large number of nodes send traffic to a particular node in an uncoordinated fashion, causing the node to become a hotspot, resulting in performance degradation.

To understand how such a behavior can occur in Pachyderm, it's important to understand the way requests are sharded in a Pachyderm cluster. Pachyderm currently employs a simple sharding scheme that shards based on file names. That is, requests pertaining to a certain file will be sent to a specific node. As a result, if you have a number of nodes processing a large dataset in parallel, it's advantageous for them to process files in a random order.

For instance, imagine that you have a dataset that contains `file_A` , `file_B` , and `file_C` , each of of which is 1TB in size. Now, each of your nodes will get a portion of each of these files. If your nodes independently start processing files in alphanumeric order, they will all start with `file_A` , causing all traffic to be sent to the node that handles `file_A` . In contrast, if your nodes process files in a random order, traffic will be distributed between three nodes.

CHAPTER 10

Getting your Results

Once you've got a few pipelines built and have data flowing through the system, it becomes incredibly important to track that flow so you can read the correct results. Let's use the *Beginner Tutorial* Fruit Stand as an example.

Here is our data flow:

Every commit of new log lines that comes into Data creates corresponding output commits on both the Filter and Sum repos. Let's say we want to programatically read the value of the file "apples" in Sum repo after each input commit. If a new commit, `master/4` , is made now, how do we know when `pachctl get-file sum master apples` will be showing us the resulting value of `master/4` and not `master/3` or `master/5` ?

To do this, we'll use a feature of Pachyderm called Provenance. We're actually only using one piece of Provenance called `flush-commit` . `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.

You can read about other advanced features of Provenance, such as data lineage, in our Advanced *Provenance* Guide, but we're just going to cover `flush-commit` here.

10.1 Using Flush-Commit

Let's demonstrate a typical workflow using `flush-commit` First, we'll make a new commit into the Data repo, `master/4` . The filter and sum pipelines (in serial) are chugging along and we want to read out the result of "apples" after the new data in `master/4` has fully propogated through our pipelines. We do this with:

```
# return the commit in Sum caused by Data/master/commit4 (<repo_name/commitID>)
$ pachctl flush-commit Data/master/4
BRANCH                           Repo/ID                                    PARENT ␣
↪     STARTED            FINISHED           SIZE
master                           Data/master/4                              <none> ␣
↪     55 seconds ago          55 seconds ago      874 B
fa59744b0e0448348159fef216f4eee9   Sum/fa59744b0e0448348159fef216f4eee9/0     <none> ␣
↪     37 seconds ago      36 seconds ago      12 B
557c7db83002419aa2634e0c0ca9f2e2   Filter/557c7db83002419aa2634e0c0ca9f2e2/0  <none> ␣
↪     46 seconds ago      37 seconds ago      200 B

# read the file
```

```
$ pachctl get-file Sum fa59744b0e0448348159fef216f4eee9/0 apple
133
```

Note: If you're manually commiting new data, monitoring jobs to wait for them to finish, and then reading the latest commit in the output repo, you don't actually need flush-commit. But as soon as you have data streaming in or want to look up a result that corresponds to a specific input commit, `flush-commit` is your answer.

Check out the API docs for ../pachctl/pachctl_flush-commit if you want a complete overview of the optional arguments.

CHAPTER 11

Updating Pipelines

During development, it's very common to update pipelines, whether it's changing your code or just cranking up the parallelism.

In the former case, changing your pipeline code completely invalidates previous results in the output repo and breaks parantage of output commits for this pipeline and all downstream pipelines. By default, Pachyderm won't delete the data from those downstream repos, instead we `archive` it so you can still access the data and we then reprocess the input data again with your new code.

Note: The update-pipeline functionality is somewhat in beta and the API will likely change a bunch in the next release. We'd love your feedback an how you're using it and what we can improve. info@pachyderm.io.

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

In some cases, such as changing parallelism, where you don't want to archive previous data and recompute results, you can pass the `-no-archive` flag.

Warning: To emphasize again, updating a pipeline without the -no-archive flag will archive ALL downstream repos and start reprocessing ALL the input commits again. 0

11.1 Archived commits

Archived commits are meant for old data that isn't relevant to the current set of pipelines but you might want to reference later.

Archived commits are nearly identical to regular commits except they don't show up on `list-commit` or `flush-commit` . You can use `list-commit -a` to see both archived and canceled commits and can still read from archived commits by referencing them by `commitID` .

You can also manually archive data:

```
# Archive all commits in all repos. This is ideal if you have a bunch of garbage␣
↪data, but want
# to keep your pipelines and repos intact. Your old data is still available using␣
↪list-commit -a.
$ pachctl archive-all
```

```
# Archive a specific commit.
$ archive-commit <commitID>
```

Note: Only use `archive-commit` if you know what you're doing, or archiving a whole branch, because it can get you into some weird situations.

CHAPTER 12

Advanced Features

This section of documentation covers some more advanced features of Pachyderm that you should understand when using Pachyderm for production data science workloads.

Provenance: Tracking data lineage, auditing data, and debugging incorrect results.

Incrementality: Optimize your cluster performance by only processes data *diffs*.

Composing Pipelines: Create and manage a complex dependency graph of pipelines.

CHAPTER 13

Provenance

We haven't had time to write full documentation on Provenance yet. It's an important feature and these docs are coming soon, we promise!

For now, our recent blog post on Provenance will give you a good overview.

If you have any questions, feel free to ask us on Slack, or email us at support@pachyderm.io and we'd be happy to help!

CHAPTER 14

Incrementality

Incrementality is an advanced feature of Pachdyerm that let's you only process the "diff" of data for each run of a pipeline. Since Pachyderm's underlying storage is version controlled and diff-aware, your processing code can take advantage of that to maximize efficiency.

Due to Math, incrementality only works on some types of computation.

14.1 Method

Incrementality is defined in the pipeline spec on a per-input basis. In other words for each input, you should specify whether you want only the new data (diff) exposed or if you want all past data available.

For each pipeline input, you may specify a "method". A method dictates exactly what happens in the pipeline when a commit comes into the input repo.

A method consists of two properties: partition unit and incrementality.

14.1.1 Partition Unit

Partition unit specifies the granularity at which input data is parallelized across containers. It can be of three values:

1. `BLOCK` : different blocks of the same file may be parelleized across containers.
2. `FILE` (Top-level Objects): the files and/or directories residing under the root directory (/) must be grouped together. For instance, if you have four files in a directory structure like:

```
/foo
/bar
/baz
   /a
   /b
```

then there are only three top-level objects, `/foo` , `/bar` , and `/baz` . `/baz/a` and `/baz/b` will always be seen by the same container but there are no guarantees about where `foo` or `bar` are processed relative to `baz` .

3. `REPO` : the entire repo. In this case, the input won't be partitioned at all and all data in the repo will be available.

14.1.2 Incrementality

Incrementality

Incrementality ("NONE", "DIFF" or "FILE") describes what data needs to be available when a new commit is made on an input repo. Namely, do you want to process *only the new data* in that commmit (the "diff"), only files with any new data ("FILE"), or does all of the data need to be reprocessed ("NONE")?

For instance, if you have a repo with the file `/foo` in commit 1 and file `/bar` in commit 2, then:

- If the input incrementality is "DIFF", the first job sees file `/foo` and the second job sees file `/bar` .
- If the input is non-incremental("NONE"), every job sees all the data. The first job sees file `/foo` and the second job sees file `/foo` *and* file `/bar` .
- "FILE" (Top-level objects) means that if any part in a file (or alternatively any file within a directory) changes, then show all the data in that file (directory). For example, you may have vendor data files in separate directories by state – the California directory contains a file for each california vendor, etc. `Incremental: "FILE"` would mean that your job will see the entire directory if at least one file in that directory has changed. If only one vendor file in the whole repo was was changed and it was in the Colorado directory, all Colorado vendor files would be present, but that's it.

Combining Partition unit and Incrementality

For convenience, we have defined aliases for the three most commonly used (and most familiar) input methods: "map", "reduce", and "global".

- A "map" (BLOCK + DIFF), for example, can partition files at the block level and jobs only need to see the new data.
- "Reduce" (FILE + NONE) as it's typically seen in Hadoop, requires all parts of a file to be seen by the same container ("FILE") and your job needs to reprocess *all* the data in the whole repo ("NONE").
- "Global" (REPO + NONE), means that the entire repo needs to be seen by *every* container. This is commonly used if you had a repo with just parameters, and every container needed to see all the parameter data and pull out the ones that are relevant to it.

They are defined below:

	Partition Unit			
Incrementality	"BLOCK"	"FILE" (Top-lvl Obj)	"REPO"	
"NONE" (non-incremental)		"reduce"	"global"	
"DIFF" (incremental)	"map"			
"FILE" (top-lvl object)				

14.2 Writing Incremental Code

Writing your analysis code to take advantage of incrementality involves understanding two ideas: What new data is available as input and what was the output last time this pipeline ran (aka: output parent commit)? To answer both of these questions, you need to understand which data is exposed where in Pachyderm.

14.2.1 Mount Path

The root mount point is at `/pfs` , which contains:

- `/pfs/input_repo` which is where you would find the latest commit from each input repo you specified. - Each input repo will be found here by name - Note: Unlike when mounting pfs locally, there is no CommitID in the path. This is because the commit will always change, and the ID isn't relevant to the processing. The commit that is exposed is configured based on the incrementality flag above.
- `/pfs/out` which is where you write any output
- `/pfs/prev` which is this pipeline's previous output, if it exists. (You can think of it as this job's output commit's parent).

The easiest way to understand how to use incrementality and `/pfs/prev` is through a simple example.

14.2.2 Example (Sum)

Sum is a great starting example for how to do processing incrementally. If your input is a list of values that is constantly having new lines appended and your output is the sum, using the previous run's results is a lot more efficient than recomputing every value every time.

First, we should set `partition: "FILE" and ``Incremental: "DIFF"` . Setting partition in this way ensures that all the values are seen by one container. If we had this set to `map` instead, we may get some input values spread across containers and we wouldn't get an accurate total. Incremental ansures that only the new values are shown.

For each run of the pipeline, `/pfs/<input_data>` will be a file with all the new values that have been added in the most recent commit. Our pipeline should simply sum up those new values and add them to the previous total in `/pfs/prev` and write that new total to `/pfs/out` .

CHAPTER 15

Composing Pipelines

Any reasonably complex analysis isn't just going be computed in a single pipeline, but instead a chain of pipelines. We often refer to chains of pipelines as dependency graph or DAG (directed acyclic graph). Before we jump into dealing with chains of pipelines, it's important to understand how pipelines deal with multiple inputs.

15.1 Multiple Inputs

A pipeline is allowed to have multiple inputs. The important thing to understand is what happens when a new commit comes into one of the input repos. In short, a pipeline processes the **cross product** of its inputs. We will use an example to illustrate.

Consider a pipeline that has two input repos: `foo` and `bar`. `foo` uses the `file/incremental` method and `bar` uses the `reduce` method.

Now let's say that the following events occur:

```
1. PUT /file-1 in commit1 in foo -- no jobs triggered
2. PUT /file-a in commit1 in bar -- triggers job1
3. PUT /file-2 in commit2 in foo -- triggers job2
4. PUT /file-b in commit2 in bar -- triggers job3
```

The first time the pipeline is triggered will be when the second event completes. This is because we need data in both repos before we can run the pipeline.

Here is a breakdown of the files that each job sees:

```
# job1 sees /pfs/foo/file-1 and /pfs/bar/file-a because those are the only files␣
↪available
job1:
   /pfs/foo/file-1
   /pfs/bar/file-a
```

```
# job2 sees /pfs/foo/file-2 and /pfs/bar/file-a because it's triggered by commit2 in␣
↪foo. foo uses an incremental input method (file/incremental)
job2:
   /pfs/foo/file-2
   /pfs/bar/file-a

# job3 sees all the files because it's triggered by commit2 in bar, and bar uses a␣
↪non-incremental input method (reduce)
job3:
   /pfs/foo/file-1
   /pfs/foo/file-2
   /pfs/bar/file-a
   /pfs/bar/file-b
```

CHAPTER 16

FAQ

Data Storage:

- *How is data storage handled in Pachyderm?*
- *What object storage backends are currently supported?*
- *What is version control for data?*
- *What are the benefits of version control for data?*
- *How do you guarantee I won't lose data in Pachyderm (i.e. replication and persistence)?*
- *How do I get data from other sources into Pachyderm?*
- *How do I get data out of Pachyderm into another system?*
- *Does Pachyderm handle data locality?*

Deployment:

- *Where/how can I deploy Pachyderm?*
- *Can I use other schedulers such as Docker Swarm or Mesos?*
- *Can I run Pachyderm locally?*

Computation:

- *What are containerized analytics?*
- *What is the data access model?*
- *What are jobs and how do they work?*
- *What are pipelines and how do they work?*
- *How does Pachyderm manage pipeline dependencies?*
- *How do I perform batched analytics in Pachyderm?*
- *How do I perform streaming analytics in Pachyderm?*
- *How is my computation parallelized?*
- *How does Pachyderm let me do incremental processing?*
- *Is there a SQL interface for Pachyderm?*

Product/Misc:

- *How does Pachyderm compare to Hadoop?*

16.1 Data Storage

16.1.1 How is data storage handled in Pachyderm?

Pachyderm stores your data in any generic object storage (S3, GSC, Ceph, etc). You can link your object storage backend to Pachyderm by following our cloud deployment guide and passing your credentials as a Kubernetes secret.

16.1.2 What object storage backends are supported?

S3 and GCS are fully supported and are the recommended backends for Pachyderm. Support for Ceph and others are coming soon! Want to help us support more storage backends? Check out the GH issue!

16.1.3 What is version control for data?

We've all used version control for code before — Pachyderm gives you the same semantics for petabytes of data. We even borrow our terminology from Git. In Pachyderm, data is organized into `repos`. If you want to add or change data in a repo, you simply `start a commit` make your changes, and then `finish` the `commit`. This will create an immutable snapshot of the data that you can reference later. Just like in Git, only the diff of the data is saved so there is no duplication. Pachyderm exposes data as a set of diffs so you can easily view how your data has changed over time, run a job over a previous view of your data, or revert to a known good state if something goes wrong. Finally, Pachyderm also let's you branch entire data sets so you can manipulate files and explore the data without affecting anyone else's work. Just like with branching in Git, Pachyderm doesn't create multiple copies of the data when you create a branch, we just store the changes you make to it.

16.1.4 What are the benefits of version control for data?

Instant revert: If something goes wrong with your data, you can immediately revert your live cluster back to a known good state.

View diffs: Analyze how your data is changing over time.

Incrementality: Only process the new data instead of recomputing everything.

Immutable data: Run analysis written today over your data from last month.

Team collaboration: Everyone can manipulate and work on the same data without stepping on each others toes.

16.1.5 How do you guarantee I won't lose data in Pachyderm (i.e. replication and persistence)?

Your data doesn't actually live in Pachyderm, it stays in object storage (S3 or GCS), so it has all the safety guarantees of those underlying systems.

16.1.6 How do I get data from other sources into Pachyderm?

Pachyderm has three main methods for getting data into the system.

1. A protobufs API that you can access through the Golang SDK. Other languages will be supported soon!
2. The pachctl CLI, which allows you to put files into Pachyderm.
3. You can mount Pachyderm locally and add files directly to the filesystem through the FUSE interface.

16.1.7 How do I get data out of Pachyderm into another system?

In addition to using the same ways you get data into the system, you can also use pipelines. Users often want to move the final results of a pipeline into another tool such as Redshift or MySQL so that it can be easily queried through BI tools. To accomplish this, it's common to add a final stage to your pipeline which reads data from Pachyderm and writes it directly to whatever other tool you want. Redshift for example, can load data directly from an S3 bucket so the last pipeline stage can just write to that specific bucket.

16.1.8 Does Pachyderm handle data locality?

Most object stores like S3 and GCS don't provide any notion of locality and so Pachyderm similarly can't provide data locality in our API. In practice, we've generally found that data locality is not a bottleneck when optimizing for performance in modern data centers.

16.2 Deployment:

16.2.1 Where/how can I deploy Pachyderm?

Once you have Kubernetes running, Pachyderm is just a one line deploy. Since Pachyderm's only dependency is Kubernetes, it can be run locally, AWS, Google Cloud, Azure, or on-prem. Check out our local installation and cloud deployment guides.

16.2.2 Can I use other schedulers such as Docker Swarm or Mesos?

Right now, Pachyderm requires Kubernetes, but we've purposely built it to be modular and work with the other schedulers as well. Swarm and Mesos support will be added in the future!

16.2.3 Can I run Pachyderm locally?

Yup! Pachyderm can be run locally using Minikube (recommended) or directly in Docker. Check out our local installation guide to get started.

16.3 Computation

16.3.1 What are containerized analytics?

Rather than thinking in terms of map or reduce jobs, Pachyderm thinks in terms of pipelines expressed within a container. To process data, you simply create a containerized program which reads and writes to the local filesystem. Since everything is packaged up in a container, pipelines are easily portable, completely isolated, and simple to monitor.

16.3.2 What is the data access model?

To process data, you simply create a containerized program which reads and writes to the local filesystem at `/pfs/...` . Pachyderm will take your container and inject data into it by way of a FUSE volume. We'll then automatically replicate your container, showing each copy a different chunk of data and processing it all in parallel.

Check out our beginner tutorial or OpenCV demo to see how this works in action!

16.3.3 What are jobs and how do they work?

A job in Pachyderm is a one-off transformation or processing of data. To run a job use the `create-job` command. In Pachyderm, jobs are meant for experimentation or exploring your data. Once you have a job that's working well and producing useful results, you can "productionize" it by turning it into a `pipeline` .

16.3.4 What are pipelines and how do they work?

Pipelines are data transformations that are "subscribed" to data changes on their input repos and automatically create jobs to process the new data as it comes in. A pipeline is defined by a JSON spec that describes one or more transformations to execute when new input data is committed. All the details of a pipeline spec are outlined in our documentation and demonstrated in our examples.

16.3.5 How does Pachyderm manage pipeline dependencies?

Dependencies for pipelines are handled explicitly in the pipeline spec. Pipelines output their results to a repo of the same name. The "input" for a pipeline can be any set of repos, either those containing raw data or one that was automatically created by another pipeline. For example, a pipeline stage called "filter" would create a repo also called "filter" where it would store the output data. A second pipeline called "sum" could have "filter" as an input. By this method Pachyderm, actually creates a DAG of data, not jobs. The full picture would look like this: raw data enters Pachyderm which triggers the "filter" pipeline. The "filter" pipeline outputs its results in a commit to the "filter" repo which triggers the "sum" pipeline. The final results would be available in the "sum" repo. Check out our Fruit Stand demo to see exactly this example.

16.3.6 How do I perform batched analytics in Pachyderm?

Batched analytics are the bread and butter of Pachyderm. Often times the first stage in a batched job is a database dump or some other large swath of new data entering the system. In Pachyderm, this would create a new commit on a repo which would trigger all your ETL and analytics pipelines for that data. One-off batched jobs can also be manually run on any data.

16.3.7 How do I perform streaming analytics in Pachyderm?

Streaming and batched jobs are done exactly the same way in Pachyderm. Creating a commit is an incredibly cheap operation so you can even make one commit per second if you want! By just changing the frequency of commits, you can seamlessly transition from a large nightly batch job down to a streaming operation processing tiny micro-batches of data.

16.3.8 How is my computation parallelized?

Both jobs and pipelines have a "paralellism" parameter as outlined in the pipeline spec. This parameter dictates how many containers Pachyderm spins up to process your data in parallel. For example, `"paralellism": 10` would create up to 10 containers that each process 1/10 of the data. Each pipeline can have a different parallelization factor, giving you fine-grain control over the utilization of your cluster. `paralellism` can be set to `0` in which case Pachyderm will set it automatically based on the size of the cluster.

16.3.9 How does Pachyderm let me do incremental processing?

Pachyderm exposes all your data in diffs, meaning we show you the new data that has been added since the last time a pipeline was run. Pachyderm will smartly only process the new data and append those results to the output from the previous run. We have extensive documentation on incrementality that'll show you the fine-grain control you can have to optimizing computation.

16.3.10 Is there a SQL interface for Pachyderm?

Not yet, but it's coming soon! If you want to query your data using SQL, you can easily create a pipeline that pushes data from Pachyderm into your favorite SQL tool such as PostGres.

16.3.11 Product/Misc

How does Pachyderm compare to Hadoop?

Pachyderm is inspired by the Hadoop ecosystem but shares no code with it. Instead, we leverage the container ecosystem to provide the broad functionality of Hadoop with the ease of use of Docker. Similar to Hadoop, Pachyderm offers virtually infinite horizontal scaling for both storage and processing power. That said, there are two bold new ideas in Pachyderm:

1. Containers as the core processing primitive — You can do analysis using any languages or libraries you want.
2. Version Control for data — We let your team collaborate effectively on data using a commit-based distributed filesystem (PFS), similar to what Git does with code.

How does Pachyderm compare to Spark?

The only strong similarity between Pachyderm and Spark is that our versioning of data is somewhat similar to how Spark uses RDD's and data frames to speed up computation. Spark is a fantastic interface for exploring your data or running queries. In our opinion, Spark is one of the best parts of the Hadoop ecosystem and in the near future, we'll be offering a connector that lets you use the Spark interface on top of Pachyderm.

What are the major use cases for Pachyderm?

Data Lake: A data lake is a place to dump and process gigantic data sets. This is where you send your nightly production database dumps, store all your raw log files and whatever other data you want. You can then process that data using any code you can put in a container. Martin Fowler has a great blog post describing data lakes.

Containerized ETL: ETL (extract, transform, load) is the process of taking raw data and turning it into a useable form for other services to ingest. ETL processes usually involve many steps forming a DAG (Directed Acyclical Graph) — pulling raw data from different sources, teasing out and structuring the useful details, and then pushing those structures into a data warehouse or BI (business intelligence) tool for querying and analysis.

Pachyderm completely manages your ETL DAG by giving you explicit control over the inputs for every stage of your pipeline. We also give you a simple API — just read and write to the local file system inside a container — so it's easy to push and pull data from a variety of sources.

Automated ML pipelines: Developing machine learning pipelines is always an iterative cycle of experimenting, training/testing, and productionizing. Pachyderm is ideally suited for exactly this type of process.

Data scientists can create jobs to explore and process data. Pachyderm will automatically let you down-sample data or develop analysis locally without having to copy any data around.

Building training/testing data sets is incredibly easy with version-controlled data. Since you have all your historical data at your fingertips, you can simply train a model on data from last week and then test it on this week's data. Getting training/testing pairs involves zero data copying or moving.

Finally, once your analysis is ready to go, you simply add your job to Pachyderm as a pipeline. Now it'll automatically run and continue updating as new data comes into the system, letting you seamlessly transition from experimentation to a productionized deployment of your new model.

Is Pachyderm enterprise production ready?

Yes! Pachyderm hit v1.2 and is ready for production use! If you need help with your deployment or just want to talk to us about the details, we'd love to hear from you on Slack or email us at support@pachyderm.io.

How does Pachyderm handle logging?

Kubernetes actually handles all the logging for us. You can use `pachctl get-logs` to get logs from your jobs. Kubernetes also comes with it's own tools for pushing those logs to whatever other services you use for log aggregation and analysis.

Does Pachyderm only work with Docker containers?

Right now yes, but that's mostly because Kubernetes doesn't yet support other runtimes. Pachyderm has no strict dependencies on Docker so we'll have support for rkt and other container formats soon.

How do I get enterprise support for Pachyderm?

If you're using Pachyderm in production or evaluating it as a potential solution, we'd love to chat with you! support@pachyderm.io

What if I find bugs or have questions about using Pachyderm?

You can submit bug reports, questions, or PR's on Github, join our users channel on Slack, or email us at support@pachyderm.io and we can help you right away.

How do I start contributing to Pachyderm?

We're thrilled to have you contribute to Pachyderm! Check out our contributor guide to see all the details. If you're not sure where to start, recent issues on Github or ones that are labeled as "noob-friendly" are good places to begin.

CHAPTER 17

Examples

17.1 Fruit Stand

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.

Fruit Stand

17.2 Open CV

Edge detection using OpenCV on whatever images you want. More interesting than the beginner tutorial, but still great for running locally in minikube and understanding intermediate-level Pachyderm topics.

Open CV

17.3 Web Scraper

Using `wget` to build a distrbuted web scraper. Scraping is such a common task that we wanted to give you a simple example just in Shell.

Web Scraper

17.4 Word Count

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

17.5 Tensor Flow

Use Tensorflow to build a neural net that analyzes Game of Thrones scripts and produces new lines for characters. This is a great advanced example if you want to learn about the intricacies of Pachyderm and ML workflows – and you get a really cool output!

Tensor Flow

CHAPTER 18

Migration

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.

To upgrade from version X to version Y, look under the directory named `migration/X-Y`. For instance, to upgrade from 1.2.2 to 1.2.3, look under `migration/1.2.2-1.2.3`.

18.1 Backup

It's paramount that you backup your data before running a migration script. While we've tested the scripts 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.

18.1.1 Backup the metadata storage system

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

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

- GCE Persistent Volume
- Elastic Block Store (EBS)

18.1.2 Backup the data storage system

We don't currently have migration scripts that touch the data storage system.

CHAPTER 19

Pachyderm File System (PFS)

Pachyderm File System - a version controlled file system for big data.

19.1 Components

PFS has 4 basic primitives:

- File
- Commit
- Repo
- Block

Each of these is simple, and understanding all of them provides a good tour of PFS.

19.2 Data as Files

Pachyderm File System (PFS) allows you to store arbitrary data in files. These files can be as large as you'd like, and store any kind of information.

We wanted to use an interface to data that is familiar to everyone. Reading/writing data to a file is as familiar as you get.

Doing this on big data sets gets interesting, but having a simple underlying interface makes interacting with the data more intuitive, and more easily accessible to developers no matter what their language of choices.

19.3 Versioning

PFS is very Git-like. A data set is compromised of many `Files` , which constitutes a `Repo` .

In PFS you version your data with `Commits` . By versioning your data, you can:

- reproduce any input or output for your processing, which in turn enables ...
- collaborating with your peers on a data set

Reproducibility and Collaboration are things we care a lot about.

We store each commit only as the data that changed from the prior commit. This is a concept borrowed from Git. Storing your data this way also allows us to enable Incrementality.

19.4 Files vs Blocks

Under the hood, we store your files in sets of `Blocks`. These are smaller (usually ~8MB) chunks of your file. By storing your data in smaller chunks, we can more efficiently read and write your data in parallel.

`Blocks` also determine the smallest indivisible chunk of your data. When performing a `map` job, each `File` is seen by multiple containers. Each container sees one or more `Blocks` of a file.

This is important because this also determines the granularity of how the data is exposed as an input. Specifically, during a `map` job, each container will see a slice of your data file. That slice will be one or more Blocks.

19.5 Block Delimiters

For certain data types (binary blobs or JSON objects), making sure that your data is divided *correctly* into indivisible chunks is important. Doing this with PFS is straightforward.

19.5.1 Default

By default, data is line delimited and a single `Block` consists of 8MBs worth of lines.

By default, the data is line delimited and stored internally as a block of no more than ~8MBs. This means that your data will never be broken up within any line.

19.5.2 JSON

For JSON data, you might have input like this:

```
{
    "foo": "bar",
    "bax": "baz"
}
{
    "foo": "cat",
    "bax": "dog"
}
```

You can see quickly how line delimiting will not work. If a block happens to terminate not at the end of a JSON object, the result during a `map` job will be a partial / invalid JSON object.

To make sure your JSON data is delimited correctly, just make sure the file in question has a `.json` suffix. This tells PFS that the data being stored is JSON, and Pachyderm will make sure each `Block` consists of whole JSON objects.

19.5.3 Binary Data

Since binary data doesn't always have a static size, and can be quite large, delimiting binary data works a bit different.

We enable this by treating every single write to that file as a separate block, no matter what the size. E.g. if you open `/pfs/out/foo.bin` and within your code write to it several times, each time you write the data will be treated as a separate block. This guarantees that a `map` job consuming your data will always see it at least at the granularity you have provided by your writes.

To require PFS to delimit blocks in this fashion, make sure your file as the `.bin` suffix.

19.6 PFS I/O

What happens when you read or write to PFS?

19.6.1 Storage

PFS is backed by an object store of your choosing (usually S3 or GCS). This allows for highly redundant consistent storage.

Each block of each of your files is content-addressed and uploaded to your object store. This gains us de-duplication of the data.

Additionally, because each commit only contains `diffs` of blocks that were written, all data stored by PFS is immutable.

19.6.2 Writing

You can never write to a Pachyderm `Repo` without making it part of a `Commit`. That means you have to *start* the `Commit`, write your data, then *finish* the `Commit`.

Here's an example:

```
$pachctl create-repo foo
$pachctl start-commit foo master
master/0
$echo 'hai' | pachctl put-file foo master/0 test.txt
$pachctl finish-commit foo master/0
# And writing in a new commit
$pachctl start-commit foo master
master/1
$echo 'bai' | pachctl put-file foo master/1 test.txt
$pachctl finish-commit foo master/1
```

In this example, we've written two words to the same file across two commits.

You'll see that writing requires a CommitID. If the `Commit` has been finished, you will only be able to read.

19.6.3 Reading

Let's try reading the file we wrote above. That would look like this:

```
$pachctl get-file foo master/1 test.txt
hai
bai
```

Notice how the output is the cumulative result of the commits.

19.7 Mounting PFS

Pachyderm uses FUSE to mount PFS. You can think of it simplistically as a network mount of PFS. While the files are truly served from object storage, you can see them locally by mounting PFS.

19.7.1 Locally

Mounting PFS locally is a great way to debug an issue, or poke around PFS to understand how it works.

To mount locally, run:

```
$ mkdir ~/pfs
$ pachctl mount ~/pfs &
```

(If `~/pfs` already exists, you may need to `umount` it first)

Now you can look around the local mount using `ls` or just point your browser at the local files:

```
# This is equivalent to `pachctl list-repo`
$ls ~/pfs
foo
# This is equiavelent to `pachctl list-commit foo`
$ls ~/pfs/foo
master/0    master/1
# This is equiavelent to a call to `pachctl get-file ...`
$cat ~/pfs/foo/master/0/test.txt
# And this is similar to `pachctl list-file ...`. It allows you to see all files in a␣
↪commit:
$ls ~/pfs/foo/master/0/
test.txt
```

Using this interface, you can grep, touch, ls, etc the files as you normally would. The exceptions are that you cannot write data to a commit that is finished.

CHAPTER 20

Pachyderm Pipeline System (PPS)

20.1 Get Started

To get started using Pipelines, refer to our Beginner Tutorial or Pipeline docs.

20.2 Overview

Pachyderm Pipeline System is a parallel, containerized analysis platform

It is designed to:

1. Write your analysis in any language of your choosing (enabling Accessibility).
2. Allow you to compose your analyses
3. Allow you to reproduce your input data, your processing step, and your output data (enabling Reproducibility)
4. Allow you to understand the Provenance of your data

20.3 Components

PPS has two components, and understand each gives you a full picture of PPS.

20.3.1 Jobs

Jobs are transformations that are only run once.

Broadly, they take the following inputs:

- a transformation image, refer to the pipeline spec for instructions on creating your own image
  - an entry point to run the transformation
  - some other configuration options about how to run the job (parallelism, partitioning method, etc)
- at least one PFS input `Repo` containing some data
  - a `Commit` ID per input repo

When creating a job, PPS:

- creates an output `Repo` with the same name as the job

- uses kubernetes to spin up containers w the image you specify, in the configuration you specify
- mounts the input `Repo` at the `Commit` specified at `/pfs/your_repo_name` for use by your code on that container
- mounts `/pfs/out` for writing output, which is connected to the newly created output `Repo`
- runs the containers with the entry point you provided
- the output is stored in a new commit on the new output `Repo`

20.3.2 Pipeline

Pipelines are configured once, but run every time new data is present in the form of a new `Commit` on any of their input `Repo` s. You can think of them as automatically up-to-date long-running jobs.

For detailed instructions on pipelines, refer to the pipeline spec

20.4 Provenance

You'll be using and composing pipelines frequently with PPS. Quickly, you're going to want to understand how your outputs are related to the inputs.

Check out the flush-commit docs for specifics on how to track provenance.

20.5 Debugging tools

Beyond provenance, your primary triaging tool is pachctl's logs. This allows you to see the log output per `Job` / `Pipeline` and debug any errors.

CHAPTER 21

Golang Client

For any Go users, we've built a Golang client so you can easily script Pachyderm commands. Check out the autogenerated godocs as a reference.

CHAPTER 22

./pachctl

22.1 Synopsis

Access the Pachyderm API.

Environment variables: ADDRESS=:, the pachd server to connect to (e.g. 127.0.0.1:30650).

22.2 Options

```
  -v, --verbose   Output verbose logs
```

22.3 SEE ALSO

- ./pachctl archive-all - Archives all commits in all repos.
- ./pachctl commit - Docs for commits.
- ./pachctl create-job - Create a new job. Returns the id of the created job.
- ./pachctl create-pipeline - Create a new pipeline.
- ./pachctl create-repo - Create a new repo.
- ./pachctl delete-all - Delete everything.
- ./pachctl delete-file - Delete a file.
- ./pachctl delete-pipeline - Delete a pipeline.
- ./pachctl delete-repo - Delete a repo.
- ./pachctl deploy - Print a kubernetes manifest for a Pachyderm cluster.
- ./pachctl file - Docs for files.
- ./pachctl finish-commit - Finish a started commit.
- ./pachctl flush-commit - Wait for all commits caused by the specified commits to finish and return them.
- ./pachctl fork-commit - Start a new commit with a given parent on a new branch.
- ./pachctl get-file - Return the contents of a file.

- ./pachctl get-logs - Return logs from a job.
- ./pachctl inspect-commit - Return info about a commit.
- ./pachctl inspect-file - Return info about a file.
- ./pachctl inspect-job - Return info about a job.
- ./pachctl inspect-pipeline - Return info about a pipeline.
- ./pachctl inspect-repo - Return info about a repo.
- ./pachctl job - Docs for jobs.
- ./pachctl list-branch - Return all branches on a repo.
- ./pachctl list-commit - Return all commits on a set of repos.
- ./pachctl list-file - Return the files in a directory.
- ./pachctl list-job - Return info about jobs.
- ./pachctl list-pipeline - Return info about all pipelines.
- ./pachctl list-repo - Return all repos.
- ./pachctl mount - Mount pfs locally. This command blocks.
- ./pachctl pipeline - Docs for pipelines.
- ./pachctl port-forward - Forward a port on the local machine to pachd. This command blocks.
- ./pachctl put-file - Put a file into the filesystem.
- ./pachctl replay-commit - Replay a number of commits onto a branch.
- ./pachctl repo - Docs for repos.
- ./pachctl run-pipeline - Run a pipeline once.
- ./pachctl squash-commit - Squash a number of commits into a single commit.
- ./pachctl start-commit - Start a new commit.
- ./pachctl start-pipeline - Restart a stopped pipeline.
- ./pachctl stop-pipeline - Stop a running pipeline.
- ./pachctl unmount - Unmount pfs.
- ./pachctl update-pipeline - Update an existing Pachyderm pipeline.
- ./pachctl version - Return version information.

22.3.1 Auto generated by spf13/cobra on 31-Oct-2016