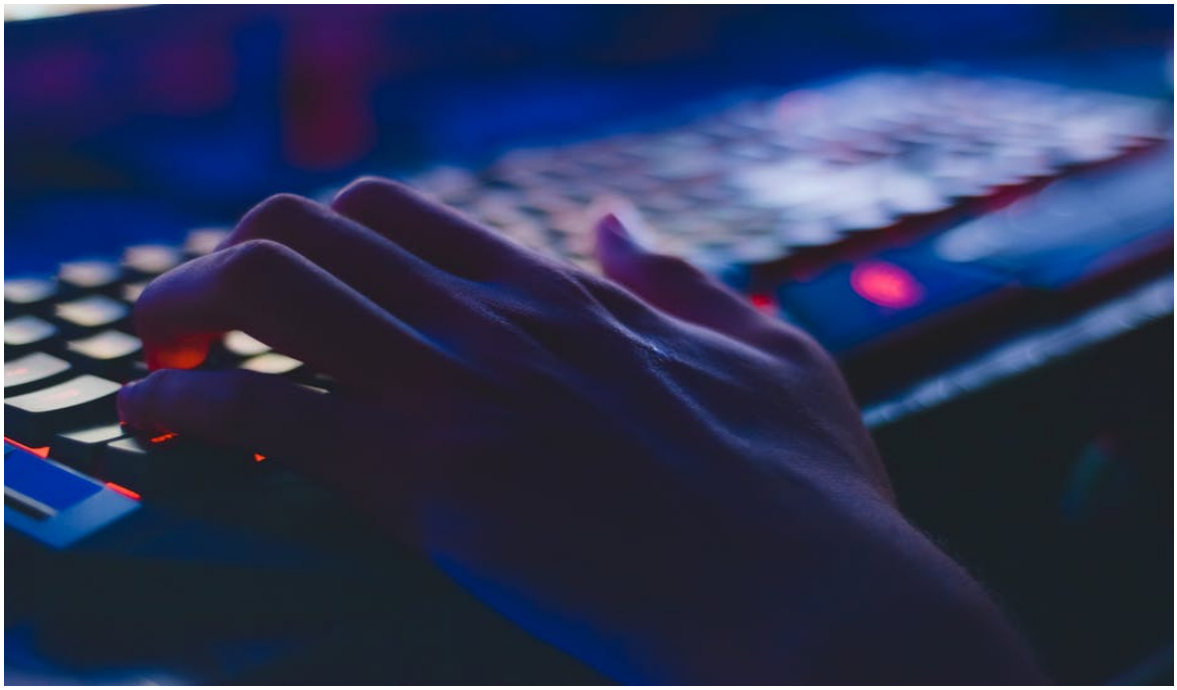
Containerized data pipelining and distributed analytics



This white paper provides an overview of production scale implementation of a data analytics solutions. We will understand the procedure for creating efficient data pipelines and automating tasks. In particular, we will explore the benefits of Docker-ized containers, Kubernetes Framework and Pachyderm. We will also explore how these frameworks actually work under the hood.

Sudeep Kurian

Table of Contents

1. Introduction

Imagine you created a really cool machine learning model while working on your organization's data. You show this work to your manager and he loves it. He gives you a thumbs up and asks you to serve this model to your clients as soon as possible. You say okay, but have no clue how to do this. How will data flow from one process to another? How will you automate batch processing sequentially with minimum external interaction? There are multiple processes involved, some need to be performed sequentially and some need to run parallelly. It seemed really easy when your manager asked you to do this but now you realise it might have a lot more to it than it seemed.

We might have often faced this issue while working in our professional environment. Fortunately many others faced the same issues and this is where the concept of data pipelining comes into play.

Data pipelining

Data pipeline can be explained as a set of different data processing elements arranged sequentially such that the output of one element becomes the input of another element and so on[1]. An example could be a basic machine learning model which follows a data pipeline as shown in the figure below. The original data source is the 'historic raw data' which might reside in some S3 object storage or some database. The data is then fed to the 'pre-process data' element. This outputs some clean data which is further fed to subsequent elements (training and testing). As we can follow, the output of one element becomes the input of the other and the whole operation of serving a machine learning model is executed by our pipeline.

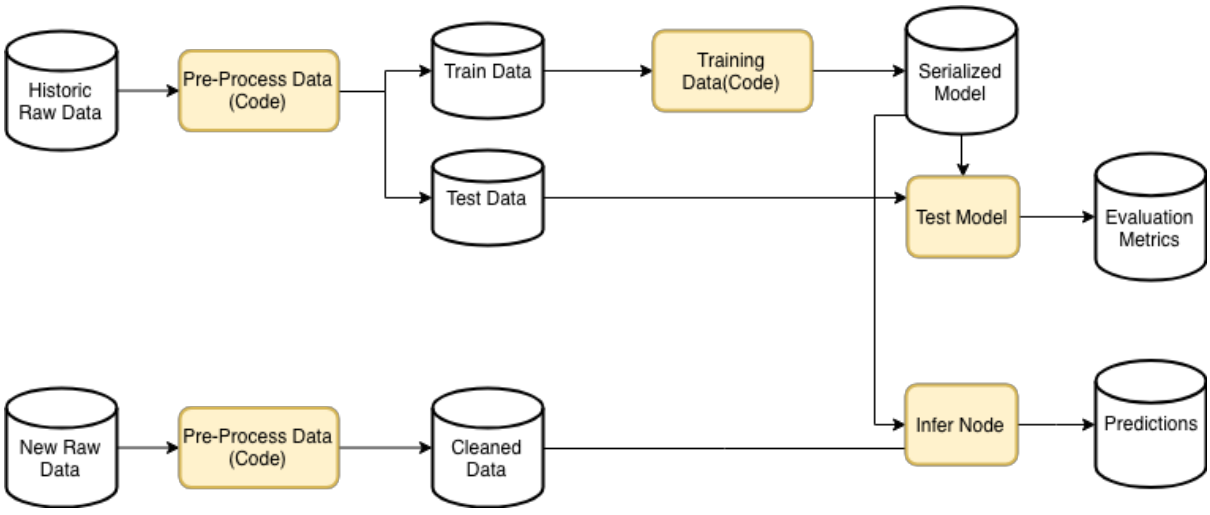


Fig 1.1: A basic machine learning pipeline

In the above pipeline, various process are to be done differently. For instance, different parts of the pipeline can get data differently. In order to serve our model results to the end user, the pipeline can use a RESTful API layer which gets input data from the end user, uses our

machine learning model on this new data, generates an output and sends the response back to the user. The model training part needs all the data together in one place. For this, we wait till we get a fixed amount of new data and then train the machine learning model on this new data (this process is also called batch processing and we will get to this later).

Following reasons explain why we would want to manage different parts of data pipelines separately.

- On one hand, the training part of the workflow is essentially run every week or maybe every month, however, the inference part of the workflow is run thousands of times per second, in other words, all the time. If we do not separate them, we will be unnecessarily using the computation power to train the model every time we need an inference.

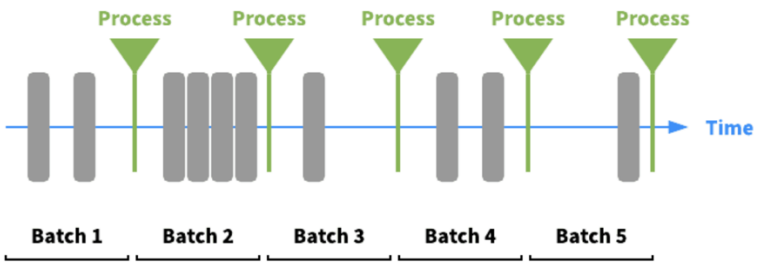
- Moreover, we might also want different pieces of our codes to be utilized at different times. We do not want to train the model as often as we want the serialized model to infer. Training a model might need a lot of time however while serving predictions to clients or stakeholders, we need a really high speed application, so having a different computing resource which trains a model and one which serves a model might be essential.

- Different parts can be managed by different teams and hence it will be very convenient if we segregate the 2 tasks, which in-turn also opens the possibility of parallel processing the two pieces, which otherwise would have been impossible.

Imagine the awesome machine learning model you created was actually a sentiment analysis model which was served as an API. This API could be used by clients on purchase. However your model is special in the sense that it uses twitter data to understand the latest 'lingo' used by millennials to improve its sentiment predictions. The model keeps taking new data to improve prediction accuracy as well as keeps up to date with the latest trends.

In the above example data flows differently in different parts of the pipeline. While predictions are served as an API where we need higher speed of processing, in order to feed new data, we can't train our model as soon as a new tweet comes out.

A good approach could be training the model once every week with the latest tweets of that week. This is known as **batch processing**. Batch pipelines are most useful for when we want to move large volumes of data at a regular interval, and we do not need to move data in real time. The figure below illustrates batch pipeline processing.

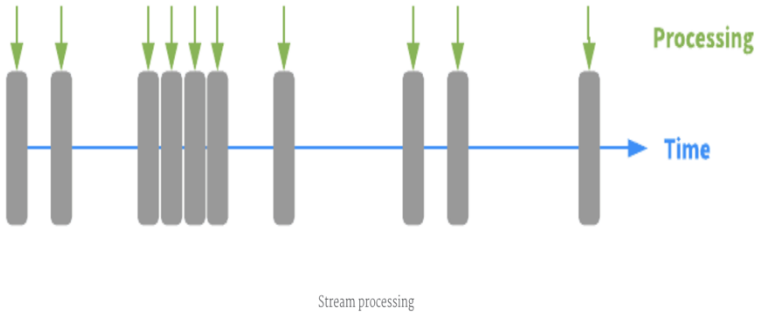


Batch processing with a time-based batch interval

Source: https://streaml.io/resources/tutorials/concepts/understanding-batch-microbatch-streaming

Figure 1.2 Batch processing

The other method of serving the model uses the **API layer pipeline** or the **stream pipeline**. In stream pipeline, each new piece of data is processed when it arrives. Unlike batch pipeline, there is no waiting until the next batch processing interval and data is processed as individual pieces rather than being processed a batch at a time. The figure below illustrates stream pipeline processing.



Stream processing

Source: https://streaml.io/resources/tutorials/concepts/understanding-batch-microbatch-streaming

Figure 1.3 Stream processing

Our current sentiment analysis model is an example where we need both API layers as well as batch processing in order to serve a mode in our pipeline. Mostly in such applications, there might be a need of both API layers and batch processing together.

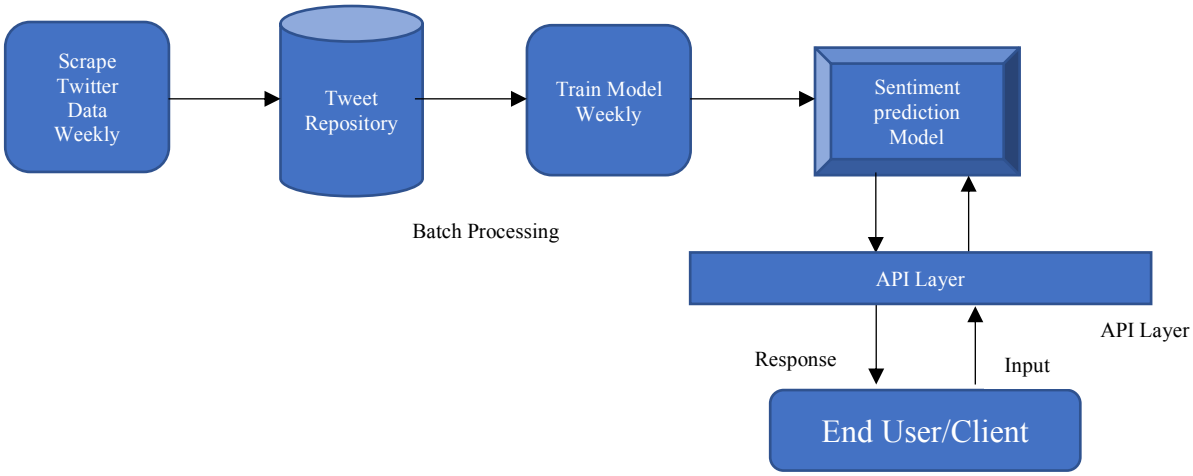


Fig 1.4 Batch and API layers in a single application

2. Containerized pipeline stages

So, now you have a basic pipeline set up for you sentiment analysis model and you you're your code written in your Jupyter notebook. However you used a new library for python. You also have a lot of other dependencies set up in your local device environment. In order to get the same reproducible results on each part of your pipeline stage upon deployment, you need to ensure each compute resource you

spin up has an environment which is set up similar to your local environment. Now that seems like a lot of work! For issues just like this, we have docker containers to the rescue.

Containerizing data pipelines with Docker

In simple words, Docker is a tool which allows developers to run their codes and deploy their applications in a sandbox environment (an environment which is isolated and is used for testing). Docker allows users to package an application with all of its dependencies into a standardized unit. It allows users to do so with the help of something called docker images and containers.

A docker image is something which includes your app and dependencies. A docker container is a running instance of a docker image. This framework gives developers the ability to create predictable environments that are isolated from rest of the applications and can be run anywhere. In other words, suppose you have your code snippets for pre-processing, training and generating sentiment predictions using the model. In order to ensure reproducibility of all these code snippets, you can create separate docker containers for each part of the whole pipeline and then deploy them.

Some of the advantage of having separate containers for each part of your pipeline are:

- If different teams create different parts of code for a pipeline, it is easier to integrate if they have been managed with Docker containers
- Docker containers are easy to set up in a cloud environment
- Majority of cloud computing providers like AWS and Google Compute Platform have embraced docker as a tool and hence it is extremely portable
- “From a security point of view, Docker ensures that applications that are running on containers are completely segregated and isolated from each other, granting you complete control over traffic flow and management of data” [2]

Creating pipeline stages

Now that we know the basic utility of docker containers and using them in our data pipeline, let’s explore how we might go about creating our pipeline stages. We shall use an example to go through the process of generating a docker image for one process and how to make it deployable in a pipeline stage.

Suppose, we want to create a deployable docker image for our pre-processing stage. We have all the code we need in our local Jupyter notebook as a start.
The figure below illustrates the process of creating a custom docker image for your application and its deployment. The pre-process code you have will be created into a Dockerfile. Docker can build images automatically by reading the instructions from a Dockerfile. Using our Dockerfile, we can generate a docker image for our code. This docker image can be versioned and pushed in a docker registry like Dockerhub which is a repository which stores different docker images in a version controlled environment. In other words, Dockerhub is just like GitHub for docker images.

Whenever an image is needed to be used, it is retrieved or “pulled” from the registry. For example, if you need to deploy your pre-processing code on a remote server, you just need to pull the docker image you created for the pre-process code and it will run exactly like it ran in your local machine.

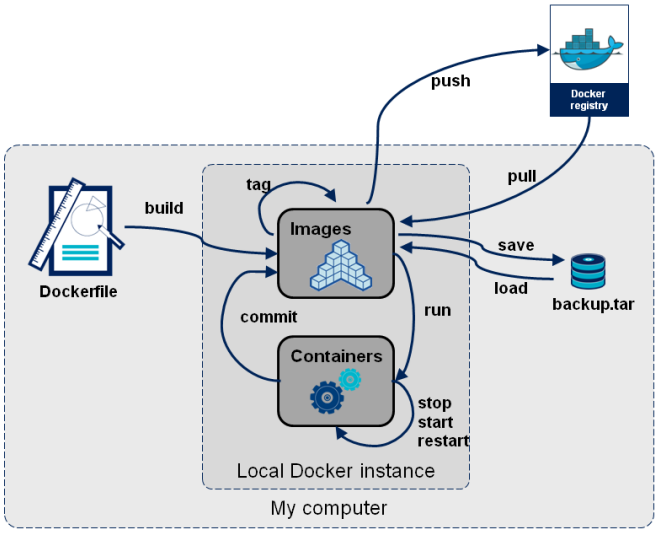


Figure 2.1 Creating a docker image and using it

Source: https://stackoverflow.com/questions/23735149/what-is-the-difference-between-a-docker-image-and-a-container

Following are the steps we need to follow to make Docker-ized containers:

- Load a docker environment on your local machine
- Create your code in the docker environment and create a docker image
- Push your docker image in a docker registry like Dockerhub
- Spin up compute resource instances like EC-2 and install docker and pull your docker image in this instance

Generally, there are 3 things that one must include in a Dockerfile -

1. Pulling Python Image - We extract the pre-existing image of python 3.6 or any of its version. We can also start from scratch and create an image and then pull it in the Dockerfile. This is the base for the dockerfile which will go on to create the final docker image.
2. Running the dependencies - We install the dependencies of the pulled image in the previous step, say we want to run multiple libraries like numpy, scikit, and pandas, so we install these dependencies and add it to the Dockerfile.
3. Adding the Code - In the final step to create an image via a dockerfile, we add the python code to out dockerfile and give it the path of the python code and it gets added to our dockerfile.

Knowing about the benefits of containerized pipeline stages, let’s say we used various Amazon EC-2 instances for each stage of your pipeline and installed docker on each instance. You pulled respective images for each part of your code and made individual containers on different EC-2 instances. What would happen if you need to scale this process up. What if there is suddenly a huge demand for your application and the traffic goes up? You might need to allocate more EC-2 resources according to the demand. How many instances would you spin up?

In relation to our twitter example, what if we have to process millions of records. We would require to spin up large number of docker-ized containers and we would require an army of people who are just logging in to their EC-2 instances and loading docker images on various instances.

You might require people to constantly view the traffic on each resource and scale it up or down on demand. There seems to be a lot of manual intervention required here. There is a possibility of a lot of human error here and following such a solution is very risky. Moreover as the scale of processing increases, after a point it is not humanly possible to go ahead with such a solution. Also it would be really expensive to implement such a solution.

Another thing we need to be concerned about is that, we might not need the entire resource of an instance for running one container. We can distribute resources of an instance across many containers. But we do not know how many of these containers can we run on one instance before running out of resources.

We need a better solution to manage all of these multiple Docker-ized containers in a much more efficient way. A solution for these issues is using a container orchestrator like Kubernetes.

3. Kubernetes for pipeline stage orchestration

Kubernetes is a *container orchestrator* which is used to manage loads across containers and instances. Orchestrating means to coordinating running a set of process across a set of resources (in our case EC 2 instances). In simple words, Kubernetes manages the various containers and balances the instances according to the computing demand for that particular container.

In order to manage containers with Kubernetes, we just have to tell it the number of containers and tasks to be run. "Kubernetes handles the nitty-gritty of rolling them out, keeping them running, and keeping the components in sync with each other."[3]

If by any chance one of our EC-2 instance goes down, it automatically spins up another instance (this feature is also known as fault tolerance). Kubernetes goes out and pulls the containers and makes sure that the state is enforced on underlying resources.

Another major advantage of using Kubernetes (henceforth called k8s) is that it enables auto scaling. "Instances need to be able to ramp up and down to suit demand, to balance incoming load, and make better use of physical resources. K8s has provisions for doing all these things in an efficient automated, hands-off way."

Hence k8s allows our pipeline stages to run on scale and helps us make the whole pipeline much more robust against failure.

Now that we know what how k8s benefits us, let's explore how we can implement a k8s framework over our EC-2 instances. There are two ways:

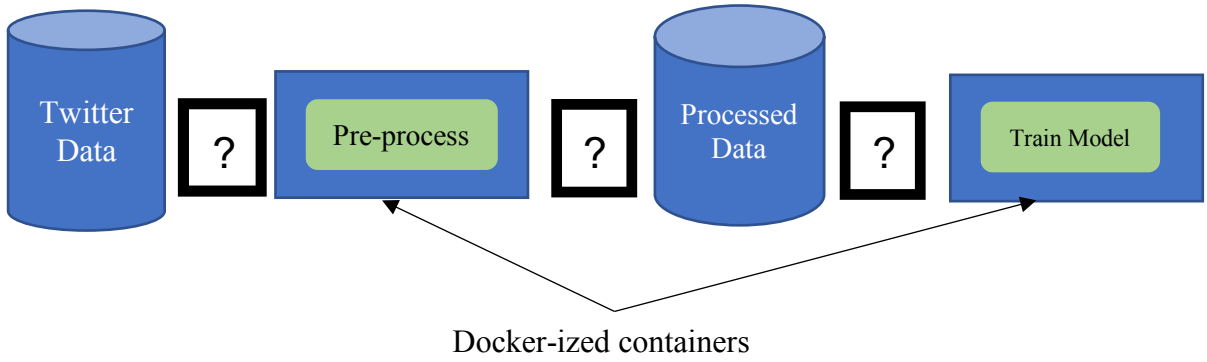
1. Roll your own solution: You can spin up EC-2 instances and install required software on these instances including the k8s framework
2. Managed services: Service providers like AWS provide managed services like the AWS Kubernetes service (EKS) where all of the setting up and spinning up instances is managed by AWS

Additional layers on top of Kubernetes

K8s offers us a lot of features which helps us to manage our docker containers and allows us to efficiently allocate resources, but it still lacks some features that we might require to manage our data pipelines.

Going back to our original twitter example, so far we enforced a k8s framework on top of our docker containers and now k8s will manage these containers and allocate resources. How will this pipeline be triggered automatically.

Kubernetes doesn't provide us with the facility of triggering a pipeline or managing the flow of data. Remember when we defined a data pipeline, we wanted data to flow from one process to another in a sequential manner. How will a batch process be triggered automatically when required twitter data is gathered periodically. How will this data be provided as an input to the pre-processing code. How will subsequent processes be executed automatically



We need some layer over k8s which fill these gaps. For this particular task we might need use the following tools:

- Amazon S3 object storage for managing and storing data
- Pachyderm layer on top of our k8s for coordinating and triggering a pipeline
- Mxnet framework for serving our machine learning model as an API

Pachyderm is basically an open source framework which allows users to create data pipelines by allowing them to trigger processes in a particular sequence. It also allows data versioning. "All of the data that flows into and out of a Pachyderm Pipeline stage is version controlled. One can look back to see what your training data looked like when a particular model was created or how their results have changed over time[4]."

Because Pachyderm utilizes software containers as the main element of data processing, we can use and combine any tools we need for a certain set of analyses or modelling. We can use codes that are developed locally on Jupyter notebooks on samples of data and utilize that exact same code in a Docker image as a stage in a distributed Pachyderm pipeline.

Pachyderm lets us easily understand the provenance of any result or intermediate piece of data easily. For example, we can quickly deduce which version of a model produced certain results and determine which training data was used to build that model.

Architectural Framework

At this point, it seems that the whole layer of our architecture is complicated and there are a lot of layers on top of each other. In order to visualise how each piece of the architecture is connected to each other, we have a graphical summary of our framework at this stage.

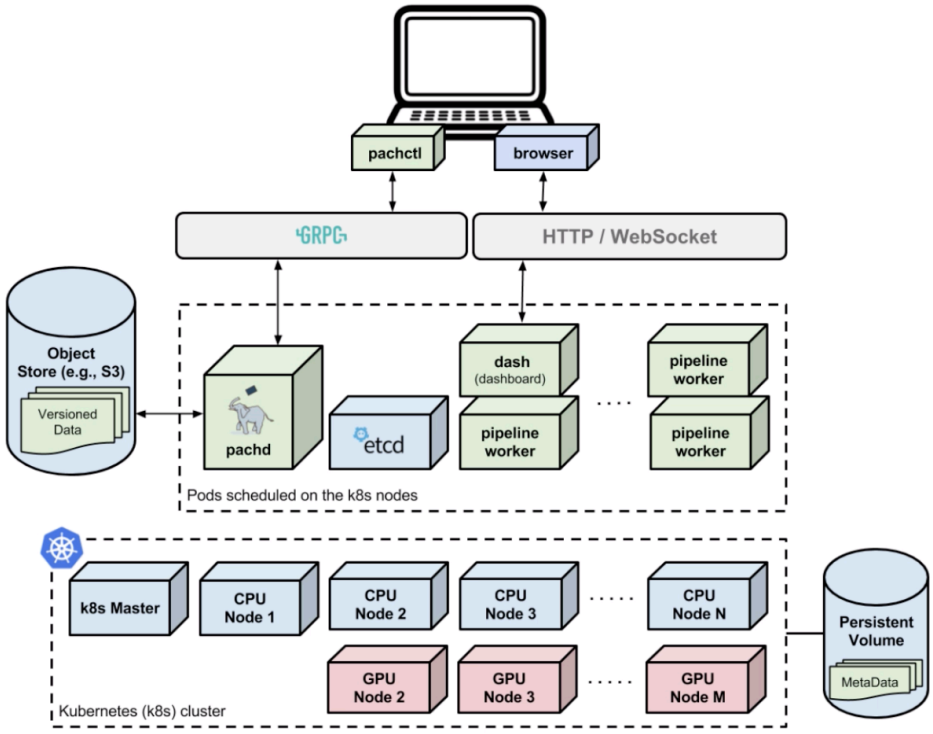


Figure 4.1 Architectural Framework of Kubernetes and Pachyderm

The bottom most, or the base layer is the Kubernetes layer, which includes multiple compute resources such as CPU, GPU, and so on. This is accompanied by a Kubernetes Master, which communicates with all resources to ensure everything is working as expected and nothing is broken.

Just above the Kubernetes layer, we have the Pachyderm at the storage layer which uses any generic Object Store like Amazon S3. We basically allocate an object store bucket to Pachyderm and behind the scenes, we constantly address the files that are put in the object store and version them simultaneously. Pipeline workers are specific pods created by Pachyderm to perform tasks specific to the pipeline, for example training or inference pipelines. The metadata information of these pods is also stored in etcd and these pods sit with k8s pods but are controlled and created by Pachyderm.

So, let's see what happens under the hood when we ask Pachyderm to create the pipeline stage for pre-processing the data which is stored as a version in the S3 object. Upon these instructions, Pachyderm will talk to the k8s API and specify the containers required for the processing and simultaneously pulls the right data from S3 and provides this data for processing. Further k8s takes up this data ans spins up Docker-ized EC-2 instances and performs the pre-processing task according to the docker image it pulls.

4. Managing a pipeline with Pachyderm and k8s

Lets check what happens when our twitter pipeline is managed by Pachyderm. How does Pachyderm manage pipelines? How is it triggering jobs ? How is data handled from one stage to another?

Pachyderm stores data in a data repository. A data repository is a collection of versioned data that Pachyderm stores in an object storage like Amazon S3 etc. We can link object storage backend to Pachyderm by following and passing credentials as a Kubernetes secret. Pachyderm holds data such that we can easily see how the data has changed over time, run a process over a previous view of data, or revert to an old version. With Pachyderm we can also create data branches for entire data sets so that we can manipulate files and explore the data without affecting anyone else's work.

The tasks handled through this include data storage, data input and output (to and from processing) and data versioning. Similar to the architecture in our local computers, when a file/commit/repo is deleted, the data is not immediately removed from the underlying storage system (e.g. S3). It is put into a read only mode where no new jobs can be created, and no data can be added.

Multi-stage pipelines can be created by repeating the following steps to build up a graph of processing steps.

- **Writing code**: Code used to process data in Pachyderm can be written using any language or library
- **Building a Docker Image**: When we create a Pachyderm we need to provide the docker images and container information that are to be run.
- **Creating a Pipeline**: We need to provide 'pipeline specifications', which are specific commands that are used to communicate with Pachyderm using JSON.

Pachyderm uses 'pipeline workers' to perform the computational parts of our pipeline, like checking if conditions for running a particular job have been met and transferring data from one pipeline element to another. Pipeline workers are just

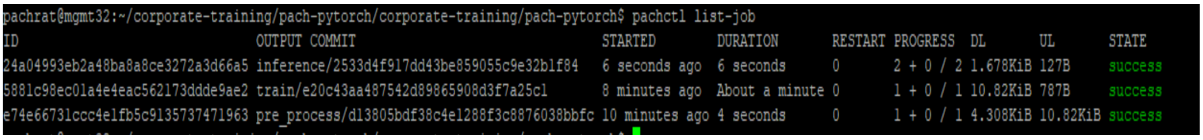
some Kubernetes pods which were actually spun up by Pachyderm to specifically work for the computational requirements of our data pipeline. When we say “job” we mean a particular task that you pipeline needs to perform. For example the model training portion of a pipeline constitutes as a “job”.

We might now be wondering, how exactly is a particular job triggered in a pipeline? Whenever we list jobs on Pachyderm, we see a number of jobs in our pipeline. Each of these jobs are triggered and carried out by one of the pipeline worker. There are certain initial conditions which are mentioned which when met automatically triggers a job to process it. For example whenever a new twitter data file is entered into the inference folder, a job gets automatically triggered to get inferences according to the trained model.

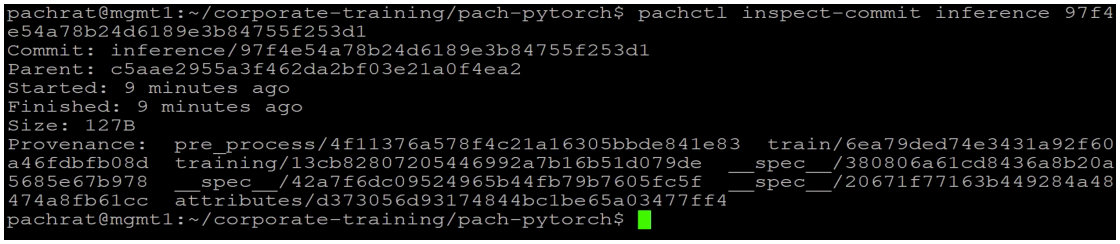
Our question now becomes, how do we feed data into Pachyderm then?
In order to do that, we need to first create input data repositories and then commit the training data set into Pachyderm. To do this, we set our working directory and then use the ‘put-file’ command in Pachyderm.

Tracking jobs in our pipeline

Now that we created our pipeline, and we have our input data inside the required data repository. How do we track what processes are occurring in the back end. Where has our pipeline stage reached?
When we use the ‘list jobs’ command on pachyderm, we can see the ID of the job and the output commit ID. The below screenshot is the response of a ‘list jobs’ command on Pachyderm. We can see exactly when a job started and the status of that particular job.

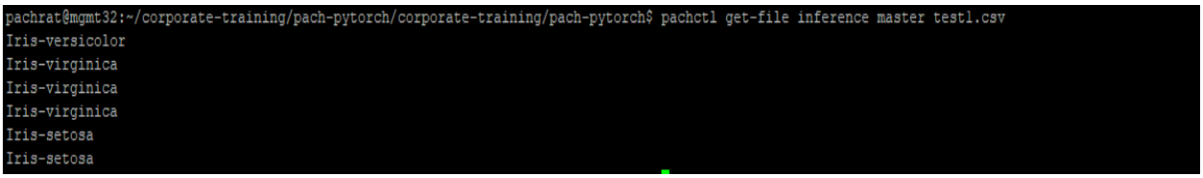


In order to track which data which created the pipeline job and what output data was created, we can use the following command:
Pachctl inspect_commit <folder name> <commit id>
This will list a history of everything which contributed to this particular result. The pachd server does this tracking



Lets assume we did train our model through the newly created pipeline and we want to see our model predictions. How do we get the inferences file out from this pipeline?

The pachctl CLI tool command get-file can be used to get versioned data out of any data repository. We can use the pachctl list-file command and flush-commit in conjunction to see what files are available and identify the latest versions respectively. The screenshot below shows an example where we use the ‘get-file’ command to see our results.



Suppose we have 7 million rows of training data from our twitter feed and we need to pre-process our data. It would make a lot of sense to parallelize this process by splitting up our data into different parts and distribute processing between various containers running the same pre-process docker image. This is called distributed processing. In order to achieve something like this in our pachyderm pipeline, we include a parameter called ‘parallelism_spec’. This tells Pachyderm how to parallelize a particular pipeline stage. In particular, let's say we want to spin up 5 pipeline workers to perform pre-processing in parallel, then we just update parallelism to 5. Pachyderm will then spin up 3 workers, each running the same pre_process.py script, to perform pre-processing in parallel.

5. Integrating a batch pipeline with APIs

We now have a complete pipeline created with a k8s cluster containing many EC-2 instances and a pachyderm layer on top which manages the pipeline. Every week our pipeline gets new tweets for the week as it’s input data and our batch pipeline gets triggered. In the end we have a new serialized model which is updated with the latest buzzwords and slangs used by millennials.

Once this model is trained using the batch data pipeline, we can update the old model and serve this model as an API using services like MXnet. We can now go back to the initial figure (figure 1.4) which explained the basic pipeline and how data would flow from one process to another. Now we can implement this model on large scale efficiently . You now have bragging rights and you can tell your manager that you saved your company a lot of money it would otherwise have spent. You deserve a raise!

6. Conclusions

In this paper, we came across various issues one might face while implementing a machine learning data pipeline on production scale. We came across a few solutions and tools we used to address to such problems and explored the benefits they provided. In particular, we saw the benefits of Docker-ized containers, Kubernetes Framework and Pachyderm. We also explored how these frameworks actually worked under the hood and how these frameworks have been made such that they complemented each other with great compatibility. Knowing about such solutions before implementing such a project on large scale can cut huge costs for any organisation.

References

[1] https://en.wikipedia.org/wiki/Pipeline_(computing)
[2] https://dzone.com/articles/top-10-benefits-of-using-docker
[3] https://www.infoworld.com/article/3268073/kubernetes/what-is-kubernetes-container-orchestration-explained.html
[4] http://Pachyderm.io/open_source.html?hsCtaTracking=e784eda0-ab38-4460-82f7-3e2364d34101%7C509343cc-7711-4536-bbf8-ff8d3057bb72