2018

Containerized data pipelining and distributed analytics

IN PRODUCTION SCALE MACHINE LEARNING

DEEPIKA JINDAL

Table of Contents

Introduction

Data Processing in an organization consists of multiple steps beginning from extracting data from various systems that can require many steps (from copying data, to moving it from an on-premise location into the cloud) to reformatting it or joining it with other data sources, preprocessing it (for reduction of unwanted features and extracting of important ones), building inferential and predictive models, and post-processing of the prediction or inferences from these models or simply extracting insights. Each of these steps needs to be done in a timely manner, and usually requires separate software's.

A data pipeline is the sum of all these steps working together in a synchronous or non-synchronous manner, and its job is to ensure that these steps are executed reliably to all data. These processes must be automated, updated as per the need of a business with a mechanism for failure revival. In production, a pipeline, also known as a data pipeline, is a set of data processing elements interlinked with each other, where the output of one or more elements is the input of the next one or more. The elements of a pipeline at the same stage are often executed in parallel or in time-sliced fashion.

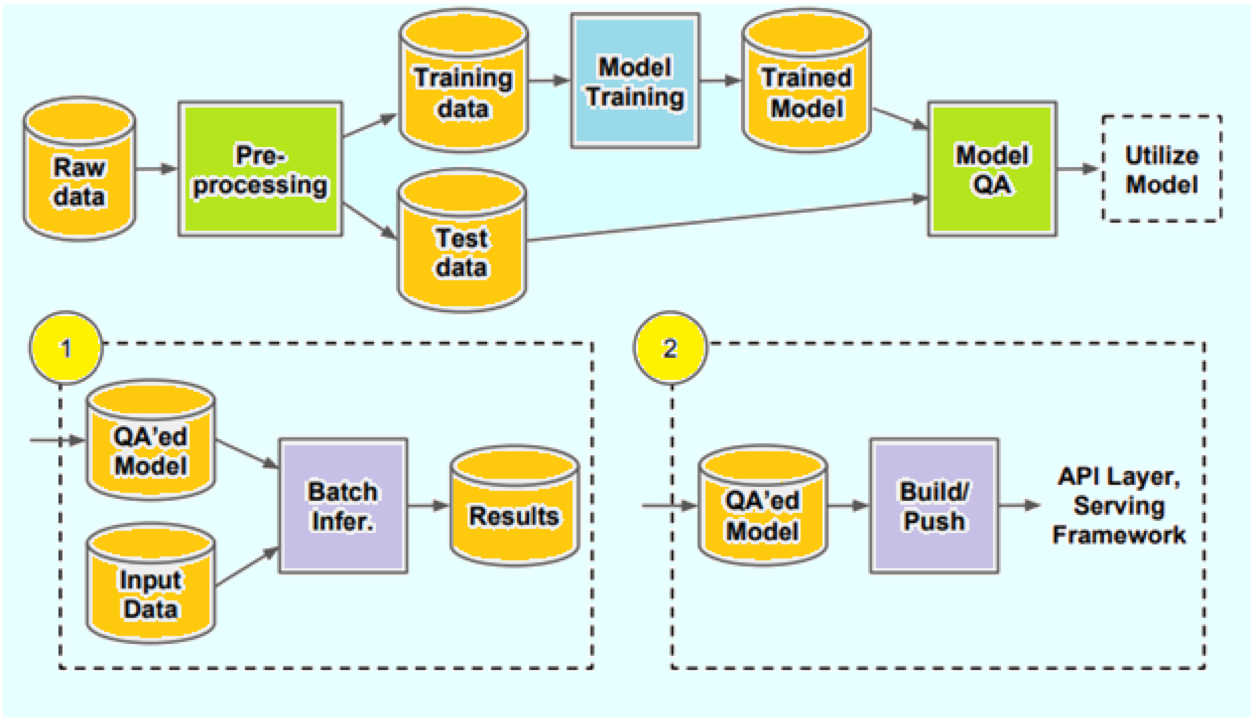
To understand the importance of production scale pipeline, lets imagine a program on Jupyter notebook, where all the data loading, cleaning and preprocessing, training, testing and prediction takes place in the same program. The issue with this kind of setup would be the limit on the amount of data that can be processed, the waiting period before getting the first set of results and then the second set of result and so on, the limit on the number of employees who can use this or work on it, multistage security, reproducibility, cost of execution and restriction on expansion. Thus, to overcome these limitations we tend to move towards or bigger, broader scale or production scale analytics.

A multi-staged data pipeline is needed to run analytics in production, and a conceptual architecture of pipeline must be defined, for parallel processing, modularity, isolation, reproducibility, individual scalability, performance and synchronized flow of data. The next section will talk about ways of processing data at various stages in a data pipeline.

Batch data pipelines

With the pace of development in technologies, there has been huge development in cloud platform. Despite the development of all these infrastructures it is still an difficult to process data real-time in a secure, inexpensive, and fool-proof manner without mis-interpretation or loss of useful information. There are generally two heralded ways of processing data today:

- Batch Processing
- Stream Processing



Difference between Batch and stream processing

Batch processing is dealing with data in non-continuous manner. It's basically splitting the data in chunks or batches and dealing it in these smaller chunks. It's fantastic at handling data sets quickly but doesn't really get near the real-time requirements. The schematic marked **one** shows batch processing is achieved. Batch data processing seems simple. Pull data from a source, apply some business logic to it, and load it for later use. When done well, automating these jobs is a huge win. It saves time and empowers decision-makers with fresh and accurate data.

Stream processing deals with processing of data in continuous manner as it is generated and is the key to turning big data into fast data. The schematic marked **two** shows stream processing is achieved. We don't need to do all our steps always.

Many steps in data processing can be done in batches, like training, wrangling and results. So, multiple parts of data pipelines are processed using batch processing. However, the parts that may need immediate processing, prediction or inferences is streamed through in the same pipeline through API streaming for example fraud detection and cybersecurity. Stream processing is the process of being able to almost instantaneously analyze data that is streaming from one device to another. We use a layer of API for stream processing where the data or feature variable are extracted real-time and directly passes in to the trained model to receive immediate inferences or outcome. For instance, a spam detector or fraud detector would need an immediate result for what category is that mail or transaction in, which would in turn lead to the prevention of that transaction

However, in stream processing begin to have issues with storage and memory and it's difficult to completely rely on stream processing.

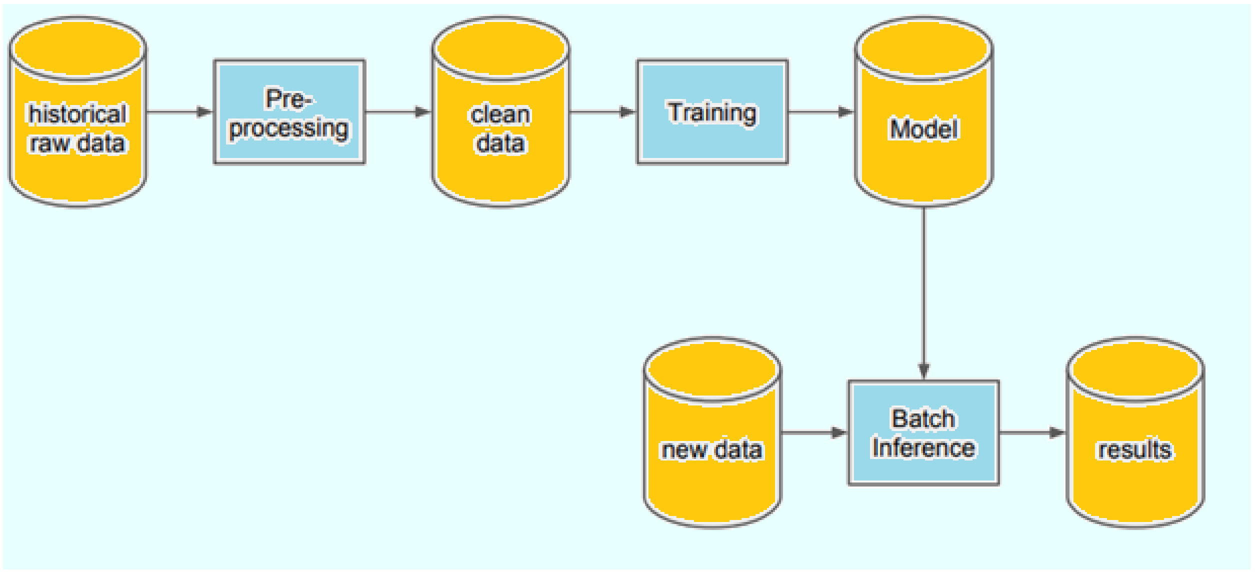
A motivating example

Data pipelines can be built in many shapes and sizes, here is a common example to get a better sense of the generic steps in the process.
Take an ecommerce system that needs to move operations data about purchases to a data warehouse. Say you've been selling laptops and accessories for some time, and we need to get a clear look at who is buying Ideapad the airbook. If we build a data pipeline to move all transactions including an Ideapad to a data warehouse, we will be able to check in on past and current Ideapads and plan for future inventory. We set out, undaunted, to build our pipeline:

- In the beginning, an order is placed, creating an order record which might include customer_id, product_ids, total paid, timestamp, and anything else that the system was built to record. Each of these items are collected into a record of the customer's action. This is the source of our data.
- Next, this data needs to be combined with data from other systems. For example, we might need to immediately combine with a customer database to verify VIP membership for free shipping. We might also want a demographics system to pick up info about shipping zip (population, median income, distance from major city, etc), or a segmentation system to associate this customer with one or more customer segments. Likely our source data will need to be combined with all these systems and possibly more. This is called joining data.
- Additional processing may be necessary. For example, some of the fields in our source data might contain discrete elements, like a zip code in an address field that needs to be accessible on its own. Further, some of the data logged by our transactional systems will be inappropriate to include in analytical systems. Specific customer information like full addresses and payment details may need to be masked.
- Now these different types of data need to be standardized. We might want to map customer age into age ranges. Color names might need to be standardized, or we might need to be sure you're consistent about how different time zones are logged.
- Finally, data can be loaded into the model building, i.e. the training and testing of a machine learning model that can help in producing insights of the expected sales, required inventory, and variations in sales. We would want to reconcile our records to be sure that all data logged by the source is accounted for in the modeling and prediction.
- We will need a system in place to notify other processes of the pipeline's completion and its final outcome, inferences or results.

- Most of these processes should be spontaneous . We can decide whether to run this process on a schedule or if it should happen continuously.

We call each of these end-to-end processes a data pipeline. So, the important parts and processes of data pipeline in interest (a machine learning data pipeline) are:



Various data pipeline stages (edited from lecture slides)

- **Sources and loading**. Data will be accessed from different sources: RDBMS, Application APIs, S3, RDS, Hadoop, NoSQL, other cloud sources, and so on. As data is accessed, security controls must be observed, and best practices are needed to be followed for reliability, consistency and performance. Data schema and data statistics are gathered about the source to facilitate pipeline design. In this example the source of our data is the operational system that a customer interacts with. This data is finally loaded into the storage.
- **Pre-processing**. On a field by field basis, data may need to be standardized in terms of units of measure, dates, attributes such as color or size, and codes related to industry standards. There may be certain features that needs to be extracted and certain other features that are not that important that needs to be dropped to reduce the size of data
- **Modeling**. Processed data is generally used in model building, training and testing of the model, that may be an inferential type, a predictive model or prescriptive model.
- **Prediction Inference**. Once the model is built, the new data is fed to these models, to get insights for the business or organization. This can be fraud prediction, anomaly detection, optimal settings from a program or in our current example prediction of requirements of goods at the warehouse.

- **Automation**. Data pipelines are normally performed many times, and typically on a schedule or continuously. Errors must be detected, and the status needs to be reported to monitoring processes.

Storing data from place to place at different stages means that different end users can query more thoroughly and accurately, rather than having to go to a myriad of different sources. Good data pipeline architecture will account for all sources of events as well as providing support for the formats and systems each event or data set should be loaded into. Updating one or multiple parts of a data pipeline is crucial. With time the prediction model may change or the amount of data or errors in processing changes, in which cases one or many of the stages in a pipeline may need to be updated as per the requirement.

Many companies have hundreds or thousands of data pipelines. Companies build each pipeline with one or more technologies, and each pipeline might be approached differently. Datasets often begin with an organization's audience or customer, but they will also originate with given departments or divisions within the organization itself. It can be useful to think of our data as events. So, events are logged and then translated across a pipeline, transformed according to the needs of our users and the systems they maintain. The following topic will talk about containers and its importance.

Containerized pipeline stages

Analytics in production scale data pipeline is computationally taxing. Moreover, each stage in the pipeline may be handled and managed by different sets of teams and would require different computational power and platforms. All the pieces of code in a data pipeline, needs to be deployed in production scale individually. However, these codes or pieces may be running on different systems, may be serving different users and domains. These domains may or may not be using same kinds of software's and systems.

1. If we use the same systems for analysis, modeling or predicting that we use for capturing data, we risk impairing both the security (at the capture end), as well as slowing down our processing thus affecting performance. For instance, a machine learning engineer would deal with a computationally heavy CPU, which would be expensive to provide to other teams. We may not want analysts, modelers to have access to production systems, or conversely, we may not want production engineers to have access to all analytics data.

2. Data from multiple systems or services sometimes needs to be combined in ways that make sense for analysis. For example, we might have one system that captures events, and another that stores user data or files. Having a separate system to govern our analytics means we can combine these data types without impacting or degrading performance.

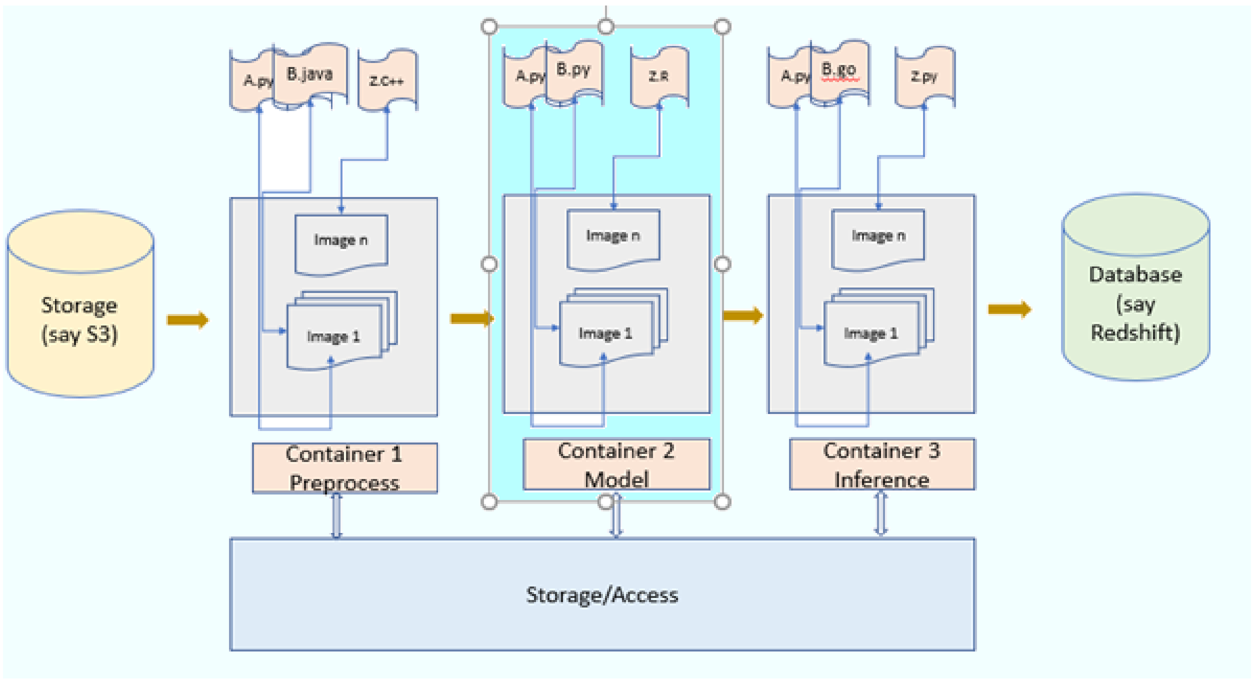
Consider an example of an ecommerce system that needs to move operations data about purchases to a data warehouse, which would need a data loading stage, a pre-processing stage, a modeling stage and a post-processing stage. This system can be taken care of by multiple teams and utilized by many others, for instance sales, finance and marketing team would access certain specific part of the data line where as data engineering, BI team would deal with others and development team would take care of certain other part of the system. We might not want the sales or finance team to interact with data collection or preprocess as this can lead to some fault in the system by some human error. Similarly, we may not want to share the market insights, next years launches or the budget for the next year to any member in the development team as this can lead to release of information and make us vulnerable.

The benefits of these isolations other that security and fault avoidance are if we need to change the way we store our data, or what we store, of any step-in processing, it's a lot less risky to make those changes on a separate system while letting the systems that back our services continue as before. Moreover, separate parts of processing may use data in different ways like training would require all data at once, whereas preprocessing can be done in smaller subsets of data. Using dockers and containerizing them takes care of the issue of isolation of various stages of pipeline and reusability of same code on various platforms at the same time addressing specific needs for various codes. Multiple dockers belonging to the same stage can be stored in one container, that ensures all parts of code at one location.

Containerization is a lightweight alternative to full machine virtualization that involves encapsulating an application in a container with its own operating environment. This provides many of the benefits of loading an application onto a virtual machine, as the application can be run on any suitable physical machine without any worries about dependencies. Docker containers are designed to run on everything from physical computers to virtual machines, bare-metal servers, OpenStack cloud clusters, public instances and more.

Creating our pipeline stages

Consider an instance of a Jupyter notebook in a PC of our analyst for processing data or collection of data in our pipeline. This notebook will be working in a CPU that would be connected to a storage. However, if this code needs to be used by many other people in an organization it needs to be on a production scale available to other teams. There may be many pieces of such codes in a data pipeline of any organization like has been discussed above. The schematic below shows multiple stages and breaks it down to the pieces that are required in a stage.



Steps in creating a pipeline stage (self-made)

These snippets are deployed in a production scale environment in form of a docker image to be reusable by different departments in a company. One or multiple images docker images would be then bound together in a docker container for integration in between dockers. For instance, a modeling stage would have model training, model selection, model validation and evaluation, which may separate snippets of code. In general, the following are the steps to follow to create various stages of data pipeline:

1. Deciding the platform to run the code on, for our case say Jupyter Notebook.
2. Understanding the requirement and implementing the piece of code that may be needed for a task in a stage, say standardizing color codes in the previous example.
3. Installing the docker engine and creating a docker image of those pieces.
4. Bucketing each of these docker images as per the requirements of the stage in a data pipeline.

Challenges in managing multiple pipeline stages

To realize the full potential of the benefits that Docker can provide, enterprise organizations need solutions designed to address these are container management challenges:

1. Multiple containers need to be managed manually, and the output from one container may be the input for other, which must be taken care of in a containerized data pipeline.
2. Manual server assignment for which container will be hosted by what server.

3. Complexity of scaling containers- for instance if in our pipeline the size of the data or the complexity of the model and thus the computation requirement changes or in general varies then it is very hard to scale them, and the pipeline flow may get affected.
4. Deploying complete applications that span across dockers and other infrastructures requires more-advanced capabilities for orchestrating deployments and managing running environments.
5. Unique monitoring requirements- Docker environments require special monitoring capabilities, for example, API-level integration with Docker, and instrumentation that is built into the Docker image.

For these reasons we need other program to take care of multiple docker-ized stages and containers in a pipeline. The following section will talk about tools that are available to overcome these challenges.

Kubernetes

Containerized pipelines have some challenges associated with them as discussed in the previous section. These challenges can be mitigated with the use of an additional layer of code for managing different containers also known as contain orchestration.

In our pipeline, orchestration is used for coordination and management of services, middleware and other complex computer systems. Historically, two methods of orchestration have been implemented:

Traditional: when orchestration is imperative and flow-based “if then, then that” kind. Which is manually building the layer of codes on the containerized pipeline which can be taxing.

New method: uses declarative orchestration when a user defines a desired state and a program tries to orchestrate that stage automatically. The orchestration tool takes care how to achieve the desired state from the current observable state. This method is commonly used for the most popular container orchestration tools, such as Kubernetes, Docker Swarm and Mesosphere Marathon. We will focus on Kubernetes in detail in this paper. Kubernetes helps us make sure that containerized applications that are integrated, run as per the requirement and helps them find the resources and tools they need to work, like in case of changes in process as mentioned in the previous section. Kubernetes is a production-ready, open source platform designed with Google's accumulated experience in container orchestration, combined with best-of-breed ideas from the community. Following are the ways in which it helps mitigate challenges.

- **Automates various manual processes:** for instance, Kubernetes will control for we which server will host the container, how it will be launched etc.
- **Interacts with several groups of containers:** Kubernetes can manage more cluster at the same time
- **Provides additional services:** as well as the management of containers, Kubernetes offers security, networking and storage services
- **Container orchestration:** Kubernetes mounts and add storage system of our choice to run apps
- **Container balancing:** Kubernetes always knows where to place containers, by calculating the “best location” for them

Features of Kubernetes

The most common container orchestration features in Kubernetes represent typical needs of containerized applications and solves the changes discussed in previous section. Here are relevant key orchestration features and how they are implemented in Kubernetes:

1) Kubernetes allows use of *Labels* for pods (group of one or more containers) and other resources (services, deployments, and so on) Then labels can be used via *Selectors* to organize and to select subsets of resources (for example, pods) to manage them as one entity. This helps in managing multiple containers at the same time just by using the label of the pod. with the same or requirement in groups reducing the implementation required of each container.

2) In Kubernetes, containers in a pod run on the same host, share an IP address and port space, and can find each other via localhost. They can also communicate with each other using standard inter-process communications like semaphores or shared memory. This helps in interaction of containers in a pod when required.

3) In Kubernetes, a *Replication Controller* manages a set of pods, making sure that the cluster is in the specified state. Specifically, it is responsible for running the specified number of pod's copies ("replicas") across the cluster to serve multiple and time changing requirements.

4) In Kubernetes, it is possible to define an *Autoscaler* for a deployment. The defined *autoscaler* will maintain the number of replicas between the specified minimal and maximal number maintaining, for example, the specified average CPU utilization across all pods so in case of impromptu increase or decrease in data or computation requirement it can upscale or downscale the replicas.

5) Kubernetes includes *cAdvisor*, an open source container resource usage and performance analysis agent that runs on each node, auto-discovers all of the containers on the node and collects CPU, memory, filesystem, and network usage statistics. *cAdvisor* also provides the overall node usage which helps in tracking the performance and active units and utilization. This helps in understanding the load and in turn improving the pipeline stages.

Shortcomings of Kubernetes

Different parts of data are managed by different containers, that are not integrated downstream, so we need extra layers on the top of kubernetes like storage. The most crucial that kubernetes lacks are:

Data Integration: Kubernetes fail to integrate various stages in a pipeline together. Kubernetes does take care of the container orchestration logic, but it fails to set up integration in various containers as the data flows downstream to various stages in the pipelines. Thus, a layer of code is needed to integrate the data flow and integrating in between various containers.

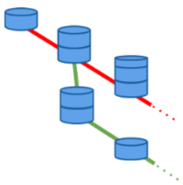
Data Management: Kubernetes cannot manage data neither does it have a specific storage unit to store models and different parts of pipeline. As such an external storage is required on top kubernetes.

Distributed Processing: Kubernetes lacks the ability to do distributed or parallel processing. An explicit piece of code is required to specify to parallelize various pods and containers in kubernetes.
The following section will introduce the tool that can address this shortcoming and talk more about pachyderms.

Pachyderm Open Source

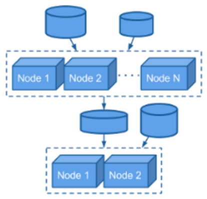
Kubernetes does a great gob at container orchestration; however, we still need to add layers on top to automate the pipeline. We can utilize a system that can serve as a platform to manage, handle and stage containers in the data pipeline. One such application that is popular these days is Pachyderm.

Pachyderm's foundational technology is open source. This open source core is designed to enable sustainable data science workflows for **data versioning** with **data pipelining**. Given below are some of the benefits of using pachyderm. (the images and content is inspired by official documentation of pachyderm)



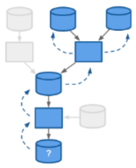
Pachyderm enables **data versioning** which is saving new copies of files/data when a change is made. This cannot be achieved in Kubernetes. Pachyderm syncs the data with latest changes and back test models on historical states of data, that helps us go back and retrieve a specific version if our file or data and even track the changes over time.

Pachyderm utilizes software containers as the main element of data processing, that enables us to use and combine any tools we may need for a certain set of analyses or modeling thereby **integrating data** and processes in various containers and stages of pipeline. Because of docker images we can just declare what processing needs to be run on what data, and Pachyderm takes care of the details. In kubernetes this was not possible.



Pachyderm automatically **parallelizes** our analyses by providing subsets of data to multiple instances of our code. We don't have to worry of explicit implementations of parallelism. This is one of most important feature of Pachyderm. It makes sure that the right data get processed by the right types of nodes, and it even auto-scale resources as our team or workloads grow or shrink.

Pachyderm lets us quickly and easily understand the **provenance** of any result or intermediate piece of data which was not possible in kubernetes. It helps us deduce the version of a model that produced certain results and determine training data for the model. This lets us iteratively build, change, and collaborate on analyses, while ensuring that we can debug, maintain, and understand those analyses over time.



Pachyderm based pipeline

A Data pipeline as discussed in our previous example constitutes of various stages form data extraction, to modeling to finally prediction and storage of results. A Pachyderm based pipeline takes care of all these stages in a systematic and integrated manner ensuring the coherence, data security and stage performance at each step. Specifications of a pachyderm data pipeline can be found in the appendix provided in this paper. Let us talk about each of the steps involved in data pipeline and how it's achieved in pachyderm.

Data Ingestion

After extraction the data that we put (or "commit") into Pachyderm ultimately lives in an object store of our choice (S3, Minio, GCS, etc.). Pachyderm addresses this data by content to build version control semantics for data which is not "human-readable" directly in the object store. However, it allows us and our pipeline stages to interact with versioned data.

Data Handling

Pachyderm allows pipelines to simply shuffle files around (e.g., organizing files into buckets). Similar to in our computer when a file/commit/repo is deleted, the data is not immediately removed from the underlying storage system (e.g. S3) for performance and architectural reasons. Garbage collection puts the cluster into a read only mode where no new jobs can be created, and no data can be added. This helps in faster movement and cleaner data pipeline.

Data management

Pachyderm exposes data as a set of diffs(differences) so we can easily view how our data has changed over time, run a job over a previous view of our data, or revert to a known good state if something goes wrong. Finally, It also allow us to branch entire data sets so we can manipulate files and explore the data without affecting anyone else's work.

Data Processing

This is a crucial part of our pipeline. In our example it is 80% of our pipeline and is done in multiple batches. Pachyderm also supports multi-stage pipelines (e.g., parsing -> modeling -> output) can be created by repeating these three steps- writing analysis code, building docker image, creating a pipeline.

Triggers

Pachyderm pipelines are triggered by changes to their input data repositories. Pachyderm Pipelines processes new data committed to their input branch as per the set triggers. For instance, in our previous example, we may want to standardize data every hour, but only want to retrain a machine learning model on that data daily since it that more time and computation resources. Triggers provides this performance benefit of deferred processing.

Job Tracking

Pachyderm uses version control kind of mechanism to process pipeline stages. As a result, each job is assigned to a commit ID. Through output commit ID we can access what are all the processes are that led to the creation of a particular result, and hence we can track what data triggered which pipeline job. The pachctl list-job command will tell us the sequence of the jobs in the pipeline. This command tells us the commit ID and from that ID we can get to know what computation and data contributed to getting.

Data Egress

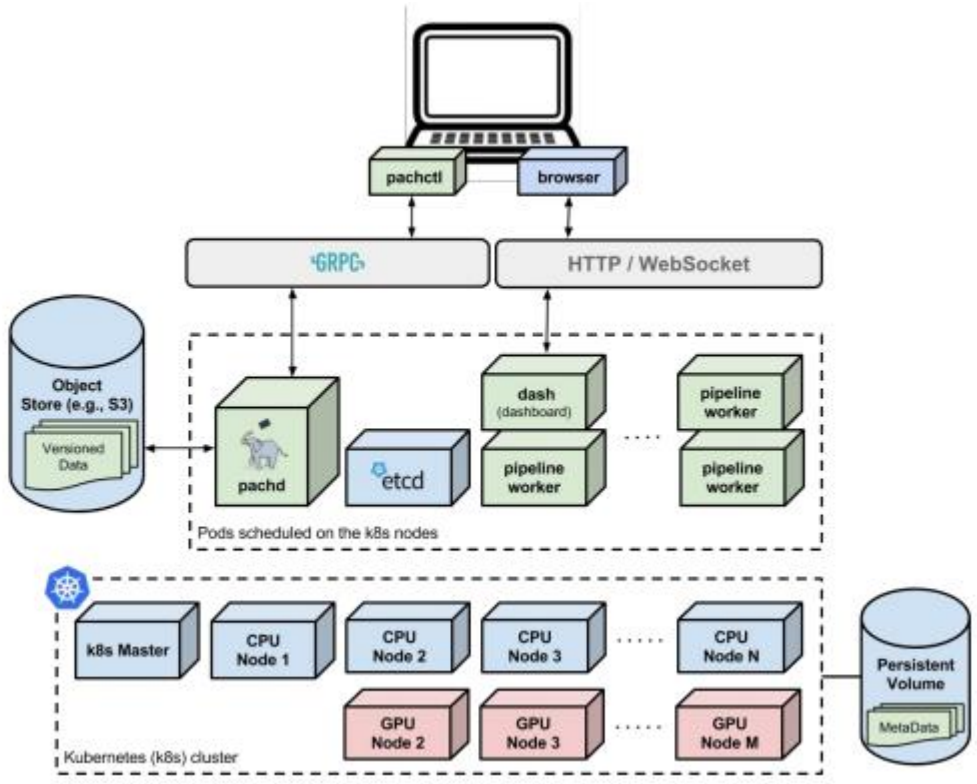
We can put data out by using commits like we did at the time of data ingestion. Alternatively, we can also use pipelines. For instance, in our example pipeline the results of a pipeline are moved to into another tool for storing results. It can be any tool such as Redshift or MySQL so that it can be easily queried. To accomplish this, a final stage can also be added to our example pipeline which reads data from Pachyderm and writes it directly to a BI tool for exploration and querying.

Processing distribution between containers

In Pachyderm pipeline workers are identical pod running the Docker image. The data that needs to be processed is spread across the various workers and made available for the code. These workers are spun up and left running to be used in the pipeline which omits the task to recreate and schedule the worker for every new job.

Layers of Architecture

In a production ready data pipeline all these are integrated to make the data pipeline functional and automatic. Let's talk about how each of these are connected in the coherence of our pipeline. We have pachyderm that is connected to various containers to integrate the data process and data flow. Each of these containers relate to kubernetes that orchestrate the containers and creates pod of each of the containers. Kubernetes manages with the master node that contains the algorithms to manage and orchestrate various pods or sets of containers. It does so by communicating through the small node of kubernetes also present in each CPU or GPU. Each CPU is a set of docker images (container) that is programmed to perform a specific task. Each container can represent a stage in a pipeline. Pachctl is used to access pachyderm. Basically, a user or an employee of the company in our previous example would access the pipeline through Pachctl which will get connected him to pachyderm and thus the pipeline or one of its stages.



Layers of architecture (from slides)

Pachyderm uses a generic object storage layer, which in our example is Amazon S3. We allocate an object store bucket to Pachyderm which it constantly integrates in various stages of pipeline and simultaneously version control the files that are put in the object store. Pipeline workers are the specific pods created by Pachyderm to perform the specific task of the pipeline, for example training or inference. 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.

Conclusion

Data pipeline constitutes of multiple stages like data extraction, preprocessing, modeling, inferences and postprocessing. There can be multiple teams and users associated with different stages of pipeline in an organization, who may be running their respective on different platforms. We need all the pieces to be reproducible irrespective of the platforms, which we find can be achieved by dockers. Various pieces in a pipeline stage are containerized to achieve successfully a cohesive code. To orchestrate various containers in an automated manner we use Kubernetes. Kubernetes creates clusters of containers called pod and allows their labeling. Kubernetes container orchestration platform provides the most sought-after orchestration features for the automated arrangement, coordination and management of containers in the clusters. However, to integrate data in pipeline stages and to achieve distributed processing we use pachyderm. Integration with pachyderm allows us parallel processing, data provenance, data versioning, data integration etc. and thus allows us to achieve an integrated data pipeline

Glossary:

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

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

Datums: Pachyderm uses the glob pattern to determine how many “datums” an input atom consists of. Datums are the unit of parallelism in Pachyderm. That is, Pachyderm attempts to process datums in parallel whenever possible.

Autoscaler: A component that automatically adjusts the size of a Kubernetes Cluster so that all pods have a place to run and there are no unneeded nodes. Works with GCP, AWS and Azure. Version 1.0 (GA) was released with kubernetes 1.8

Pods: A Kubernetes pod is a group of containers that are deployed together on the same host. If you frequently deploy single containers, you can generally replace the word "pod" with "container" and accurately understand the concept.

Caadvisors: cAdvisor is an open source container resource usage collector. It is purpose built for containers and supports Docker containers natively. Unlike most elements within Kubernetes that operate at the Pod level,cAdvisor operates per node.

Appendix

A pipeline requires all the following required fields and may have the optional depenting on the requirements.

1. Name (required) pipeline.name is the name of the pipeline that you are creating. Each pipeline needs to have a unique name. Pipeline names must: contain only alphanumeric characters, _ and - ,begin or end with only alphanumeric characters (not _ or -), and be no more than 50 characters in length

2. Input (required) input specifies repos that will be visible to the jobs during runtime. Commits to these repos will automatically trigger the pipeline to create new jobs to process them. Input is a recursive type, there are multiple different kinds of inputs which can be combined together. The input object is a container for the different input types with a field for each, only one of these fields be set for any instantiation of the object.

3. Description (optional) description is an optional text field where you can put documentation about the pipeline.

4. Transform (required)

5. Parallelism Spec (optional) parallelism_spec describes how Pachyderm should parallelize your pipeline. Currently, Pachyderm has two parallelism strategies: constant and coefficient.

6. Resource Requests (optional) resource_requests describes the amount of resources you expect the workers for a given pipeline to consume. Knowing this in advance lets us schedule big jobs on separate machines, so that they don't conflict and either slow down or die.

7. Resource Limits (optional) resource_limits describes the upper threshold of allowed resources a given worker can consume. If a worker exceeds this value, it will be evicted.

8. Datum Timeout (optional) datum_timeout is a string (e.g. 1s, 5m, or 15h) that determines the maximum execution time allowed per datum.

9. Datum Tries (optional) datum_tries is a int (e.g. 1, 2, or 3) that determines the number of retries that a job should attempt given failure was observed.

10. Job Timeout (optional) job_timeout is a string (e.g. 1s, 5m, or 15h) that determines the maximum execution time allowed for a job.

11. Output Branch (optional) This is the branch where the pipeline outputs new commits. By default, it's "master".

12. Egress (optional) egress allows you to push the results of a Pipeline to an external data store such as s3, Google Cloud Storage or Azure Storage. Data will be pushed after the user code has finished running but before the job is marked as successful.

13. Standby (optional) standby indicates that the pipeline should be put into "standby" when there's no data for it to process.

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

15. Cache Size (optional) cache_size controls how much cache a pipeline worker uses.

16. Enable Stats (optional)enable_stats turns on stat tracking for the pipeline. This will cause the pipeline to commit to a second branch in its output repo called "stats".

17. Service (alpha feature, optional) service specifies that the pipeline should be treated as a long running service rather than a data transformation.

18. Max Queue Size (optional) max_queue_size specifies that maximum number of elements that a worker should hold in its processing queue at a given time.

19. Chunk Spec (optional) chunk_spec specifies how a pipeline should chunk its datums.

20. Scheduling Spec (optional) scheduling_spec specifies how the pods for a pipeline should be scheduled.

21. Pod Spec (optional) pod_spec is an advanced option that allows you to set fields in the pod spec that haven't been explicitly exposed in the rest of the pipeline spec.

References

1) https://www.dremio.com/what-is-a-data-pipeline/
2) https://dzone.com/articles/batch-vs-stream-processing-which-should-you-choose
3) https://www.dremio.com/what-is-a-data-pipeline/
4) https://rancher.com/top-5-challenges-with-deploying-container-in-production/
5) http://www.cms.lk/introduction-to-kubernetes/
6) https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes
7) http://docs.pachyderm.io/en/latest/enterprise/overview.html
8) And lecture slides