Containerized data pipelining and distributed analytics

By

Sushil Ram Achamwad

When data becomes an indispensable asset for a business, its efficient management is of paramount importance. Every day business operations such as serving products, managing internal / external communications, reaching out to customers through marketing, taking decisions by analyzing business performance, auditing, data maintenance etc. require good deal of data interaction. A data pipeline is what that makes this interaction feasible. Data pipeline is essentially a set of automated actions which enable seamless exchange and processing of data residing in remote data systems. Actions in a data pipeline are often automated such that output of one action becomes input for the next one. This way a great deal of manual intervention in data processing is reduced.

Below is an example of a broad level data pipeline:

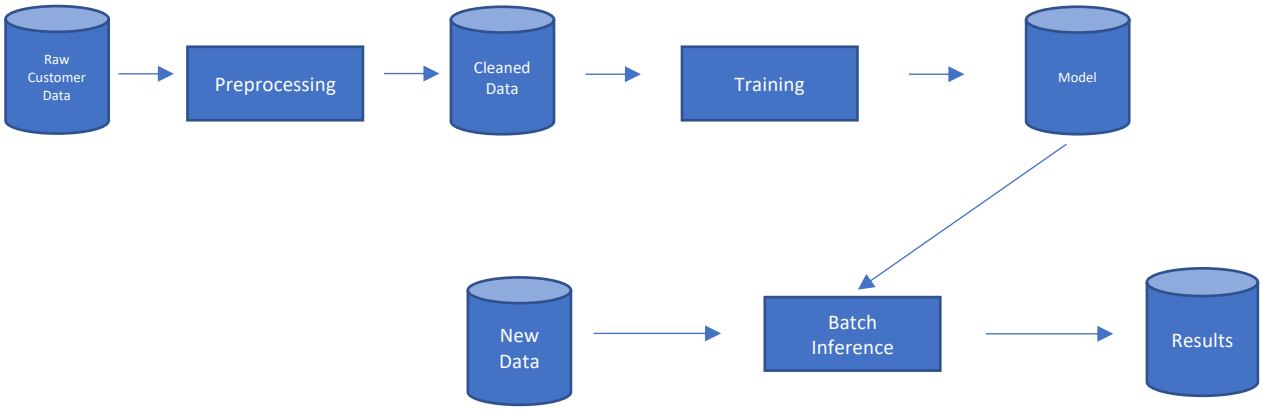

Fig. 1 Example of Data Pipeline

*source: Prof. Whitenack lecture slides

Broadly there are two types of data processing methods, batch data processing and real time data processing. In real time data processing, required actions of the data pipeline are executed real time without interruption with continual input, processing and output of data. Whereas, in batch processing data is processed in prescheduled batches and not real time. A data pipeline may comprise of both of these processes. In typical data analytics pipeline, one may not require training a model every time after additional training data is available. So, the actions such as “preprocessing” or “training” can be batch processed at scheduled intervals defined by user (ex. weekly, monthly etc.) If one requires the model results for every additional input data, these actions need to be processed real time in the data pipeline.

There may be cases where, we may not need real time preprocessing and training of the model however we may require the model to give real time results for any additional test data. In such cases we can batch process the “preprocessing” and “training” steps and process the “inference” step real time with supporting API’s.

One can understand batch data pipeline from pipeline in Fig.1. In this data pipeline there are three key processes involved, preprocessing raw input data, training a model with preprocessed data and generate inferences for new data. Here the user may schedule running the preprocessing and training code weekly. That means, results of the training model which for example serialized in ONNX file will not change for a week. During this period even though, additional raw data may

have been added daily, the inference of new test data will be based upon model generated with old dataset.

Many organizations manage batch data processing for different purposes. For example, credit card companies decide credit limits of each individual customer based on machine learning models. These models take input of customer information, preprocess the data to remove any inconsistencies, train statistical models that predict credit limits so that the customer doesn't default on payments. Here, the company may train their model every weekend even though they might be aggregating additional input data for training every day. So, here, the training stage of the pipeline will be batch processed weekly.

Containerized pipeline stages

In earlier days, we used to install all the operating systems or softwares required to run an application. Later on, technology of virtual machines became popular which enabled us to virtually run different operating systems on a local machine. However, these VM's often occupied large space in the system and were quite difficult to manage from resources point of view. These disadvantages of virtual machines gave rise to the development of docker.

Docker makes management and deployment easier by building containers. Containers allows developer to package application with all their required parts such as libraries and other dependencies. By making container it becomes easier and more dependable to run the package on the Linux operating system even though the machine may have different customization and setting from the machine that was used for writing and testing the code. Since package is installed at once the amount of time required for installing the pre-requisites is drastically reduced.

Its beneficial to make the stages of our data pipeline containerized with Docker because it facilitates standardized, simple and faster data configurations. By containerization, each individual step can be executed independently. It also makes it feasible to modify or update these steps individually so that entire data pipeline is not affected by it. When we have "training" and "inference" steps of the pipeline independently containerized we can make changes to the "training" step without affecting "inference" step which may be using real time data processing on constantly updated test data. Independent containerization also gives a flexibility to scale each of the data pipeline process independently in terms of resources (ex. No of compute nodes like EC2 in AWS). It enables distributed processing at feasible data pipeline stages. For example, "preprocessing" of data can be distributed between different computing systems. However, "training" stage of the pipeline demands all the preprocessed data to come together to execute hence is not feasible for distributed processing.

Now, let's understand how we can create pipeline stages. Typically creating pipeline stages involves developing each stage of the data pipeline at a local system and then deploying them to cloud services such as AWS. Below steps are usually involved in create pipeline stages:

- Get raw input data. This data can be in user defined format like "csv".
- Preprocess input data using a code which removes inconsistencies and converts the raw data to a standardized format that can be used in training stage
- Develop a Training model. This step may involve analyzing model performance using different statistical techniques (Ex. Linear regression, neural networks etc.). These models can be developed using different coding languages such as R and Python.
- Select the best performing training model
- Run the model with testing data and generate results

- Create docker files for executing each of the computing steps such as preprocessing of data, training of data and generating model inference. These docker files will be consisting of docker images for each of the code to be executed
- Create storage resources on cloud infrastructure such as Amazon S3 that will be storing input data
- Spin up all the compute resources required to run the model (eg. EC2). Run docker engine on each of the EC2 instances.
- Run docker containers for each of the computing stages.

Often times, managing multiple Docker-ized pipeline stages can become challenging. Main disadvantage of individually spinning each EC2 instance for running a container that's processing a data pipeline increases manual intervention and in turn scope of error. In production scale, when a step such as "preprocessing" is executed in distributed way, a docker file needs to be run on each distributed system manually and sometimes this number can be in thousands and hence practically impossible to execute physically. Even when distributed processing is not used, if number of stages in a data pipeline change as per requirement then running and coordinating each of the docker run becomes problematic.

Kubernetes for pipeline stage orchestration

Now, coming from computing efficiency point of view, one docker container may not use all the computing resources (e.g. EC2). So, its efficient to run multiple docker containers on each instance. But then how can we know the number of containers to run at each instance so that they do not exhaust the capacity of a computing resource. This operation is made feasible with container orchestration which essentially orchestrates/conducts/manages use of different computing resources for containers. One of the famous container orchestrators is Kubernetes.

In AWS data pipeline, each instantiated EC2 instance utilizing Kubernetes has Kubernetes running along with docker engines. One of the instances will have Kubernetes API running on it. This API interacts and manages another Kubernetes running on the cluster. This way Kubernetes API knows if a compute resource runs out of capacity which enables it to efficiently manage running containers.

Sample Container distribution (Kubernetes API running of EC2 Resoure 1				
All instances running	Containers	C1	C2	C3
	EC2 Resource	**1**	**2**	**3**
		K8 API		
Node 3 Down	Containers	C1	C2, C3	
	EC2 Resource	**1**	**2**	**3**
		K8 API		

Fig.2 Kubernetes Container Distribution

One of the key advantages of Kubernetes is that, if one of the compute nodes goes down it reallocates the container to another running compute resource. This can be visualized from above Fig.2. Here, assume that there are three docker containers running in the pipeline. Each of the docker container is running on a different EC2 resource instantiated on AWS namely, resource 1,

2, and 3. To efficiently manage containers Kubernetes are installed in each EC2 resources. Kubernetes API is running on EC2 Resource 1 which is managing other Kubernetes nodes. So, when Resource 3 goes down, Kubernetes automatically allocates resource 2 for running the container assuming that resource 2 has enough capacity to run both the containers.

Further, Kubernetes reduces manual intervention in running the docker containers significantly. It helps in better management and coordination in running containers. It helps in efficient capacity utilization and resource allocation. Kubernetes have a unique feature of Autoscaling resources: meaning, in case where all resources get full, the Kubernetes have the power to spin up another EC2 instance or compute resources as per requirement.

Additional layers on top of Kubernetes

Kubernetes are powerful in managing operations for running containers, but it have a distinct disadvantage of not being able to coordinate with data pipeline. Although it enables us to do distributed processing, it doesn't have any built-in functionality to do distributed or parallel processing also known as data shorting.

Though autoscaling function of Kubernetes enables user to run more containers across our EC2 cluster, it doesn't have any concept of data management, so they can't command which container to use which data.

To cover above shortcomings of Kubernetes we require two additional layers on top of Kubernetes:

1. Pachyderm:
    a. Pachyderm lets the user deploy and manage multi stage pipeline. It overcomes the issue of Kubernetes not being able to interact with data.
    b. Pachyderm provides data versioning feature which enables the user to deploy the pipeline in a version-controlled way: meaning keep track of historical data at each pipeline stage

2. Amazon S3:
    a. It is a Storage resource provided by AWS. S3 enables data versioning via Pachyderm

Managing a pipeline with Pachyderm and k8s

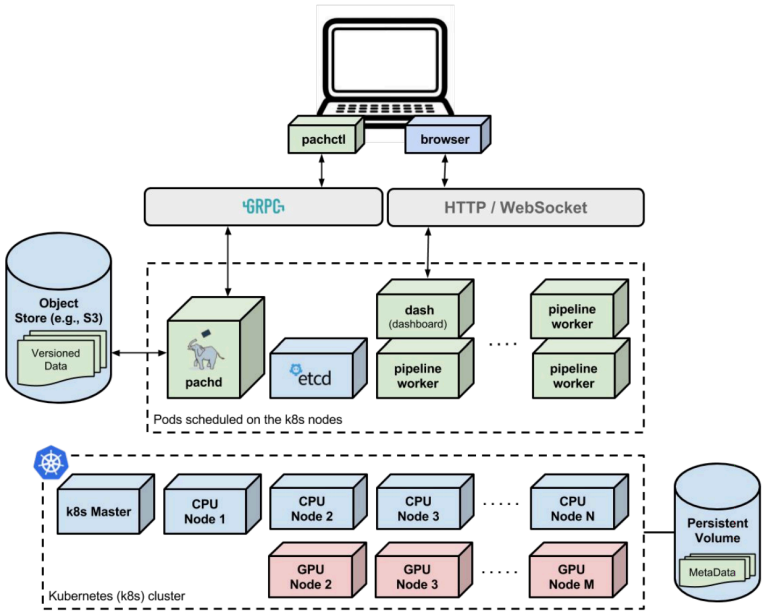

Fig. 3 Above schematic explains different layers of architecture for deploying the pipeline.

*source: Prof. Whitenack lecture slides

As shown in Fig 3, in k8s/pachyderm-based pipeline, four types of pods are created by Pachyderm on top of k8's layer namely Pachd, etcd, dash and Pipeline workers to execute predefined tasks.

Here, Pachd is used to manage data and pipeline. Pachd knows which containers to run in different stages of pipeline and commands Kubernetes to run those accordingly. Pachd stores the metadata in "etcd" that includes information about what jobs are running, what pipelines are to be executed, what data repositories exist etc. We direct Pachd to create repositories of data for each of the steps in data pipeline. Then we try to figure out how to spin up various data processing steps and their output if successfully processed. Pachd manages all the data associated with processes in data pipeline in a versioned way. So, it creates repositories of data and stores versions of data in those repositories. This data is automatically stored in S3 buckets. The output is stored in repository created for each step of data pipeline.

Pachyderm layer also has Pipeline workers which are basically Kubernetes pods spun up by Pachd to do computational part of the pipeline such as preprocessing, training or batch inferencing. Each of these pipeline process will correspond to at least one pipeline worker that's running on Kubernetes to execute that particular task.

In pachyderm, pipelines are triggered based on predefined logic specified in pipeline information. This step is automatic: meaning if the logic specifies that training stage of the pipeline should be

triggered when there is preprocessed data in training stage repository then whenever data is added in training repository, the training pipeline is executed.

Data versioning is one of the most powerful features of Pachyderm as often we require to know how the input data has changed or how the performance of model changed with time. Data versioning becomes possible with Pachyderm because it tracks every change in each of the repositories of the pipeline and assigns each change an ID through which all actions executed due to that change can be tracked for the entire pipeline.

Another powerful feature of Pachyderm is that it makes parallel or distributed processing possible. With distributed processing, different EC2 instances are spin up with corresponding number of Kubernetes containers to parallelly execute a specific pipeline. Pachyderm makes this step very simple. To perform distributed processing on a specific stage of pipeline, user only needs to specify no of parallel systems to be spun up in “parallelism spec” constant value in pipeline information. Pachyderm immediately takes this input and distributes the process between specified number of containers.

Integrating a batch pipeline with APIs

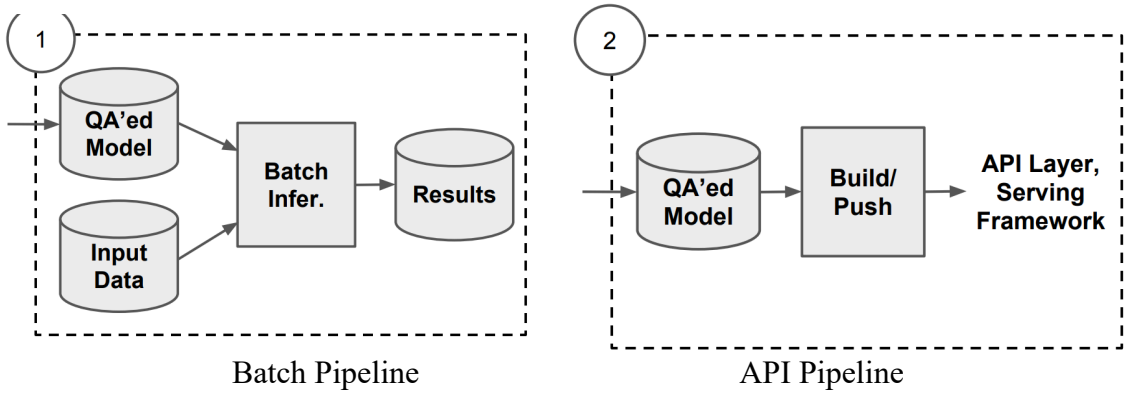

Fig.4

*source: Prof. Whitenack lecture slides

Fig.4 explains the difference between Batch and API pipeline. In real time processing, API layer utilizes serialized models to get inference in real time whereas in batch processing the serialized model is generated as per a user defined schedule. Implementation of both data processes is dependent upon the nature of application. For example, if a stock prediction model is developed by a company then it makes more sense to deploy real time API processing pipeline than a batch pipeline because the data constantly changes, and a prediction model trained on older datasets have limited value in financial markets.

Conclusions

This study started by discussing how data pipelines have become integral part of data management in recent years. We discussed about how we deploy batch and real time/API pipelines as per business requirements. Often, we use a combination of these pipelines in our applications. Containerization are found to have contributed immensely in development of data pipeline in cloud infrastructure. Further the development of Kubernetes and Pachyderm have enhanced the application of data pipeline by reducing manual intervention in running sequential data processes and enabling distributed processing that overcomes the problem of computational capacity significantly. Going forward, despite limitation of individual technologies like Kubernates, collectively, data pipelines seem to have a promising future in cloud applications.